Support des applications distribuées dans les réseaux mobiles ad hoc à continuité partielle

Abdulkader Benchi

THÈSE / UNIVERSITÉ DE BRETAGNE SUD
UFR Sciences et Sciences de l'Ingénieur
sous le sceau de l'Université Européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : Informatique
École Doctorale SICMA

présentée par

Abdulkader BENCHI

IRISA Institut de Recherche en Informatique
et Systèmes Aléatoires

Middleware Systems for Opportunistic Computing in Challenged Wireless Networks

Thèse soutenue le 29-06-2015,
devant la commission d'examen composée de :

M. Didier DONSEZ
Professeur des Universités, Université Joseph Fourier-Grenoble 1 / Président

M. Sergi ROBLES
Associate Professor, Universitat Autònoma de Barcelona / Rapporteur

M. Frédéric WEIS
Maître de Conferences (HDR), Université de Rennes 1 / Rapporteur

M. Frédéric GUIDEC
Professeur des Universités, Université de Bretagne-Sud / Directeur de thèse

Mme. Pascale LAUNAY
Maître de Conferences, Université de Bretagne-Sud / Encadrant de thèse

Abstract

OPPORTUNISTIC networks (OppNets) constitute an appealing solution to complement fixed network infrastructures –or make up for the lack thereof– in challenged areas. Researches in the last few years have mostly addressed the problem of supporting networking in OppNets, yet this can only be a first step towards getting real benefit from these networks. Opportunistic computing goes beyond the concept of opportunistic networking, and provides a new paradigm to enable collaborative computing tasks in such environments.

In the realm of opportunistic computing, properly designing, implementing and deploying distributed applications are important tasks. An OppNet-dedicated application must be able to operate and maintain an acceptable level of service while addressing the many problems that can occur in these networks, such as disconnections, partitioning, long transmission delays, transmission failures, resource constraints, frequent changes in topology, and heterogeneous devices.

Much of the complexity and cost of building OppNet-dedicated applications can be alleviated by the use of high-level programming models. Such models can be supported by middleware systems capable of transparently addressing all the above-mentioned problems.

The work reported in this dissertation focused on providing insight into the fundamental problems posed by OppNets, so as to analyze and solve the problems faced by application developers while dealing with these environments. The research focused on identifying well-known high-level programming models that can be satisfactorily implemented for OppNets, and that can prove useful for application developers. In order to demonstrate the feasibility of application development for OppNets, while assessing the benefits brought about by carefully designed middleware systems, a couple of such systems have been designed, implemented, and evaluated as part of this work.

These middleware systems respectively support distributed messaging (through message queues and topics), the tuple-space model, and consensus solving in OppNets. They are supplemented with fully-functional implementations, that can be used in real settings, and that are all distributed under the terms of the GNU General Public License (GPL). Real-life experiments and simulations have been realized so as to evaluate the effectiveness and efficiency of these systems in real conditions.

Résumé

LES réseaux mobiles opportunistes (ou OppNets, pour Opportunistic Networks) constituent une solution séduisante pour compléter les réseaux fixes d'infrastructure, voire compenser leur absence dans des zones sinistrées ou défavorisées.

Les recherches menées ces dernières années ont principalement visé à permettre les transmissions dans les OppNets, mais ceci ne peut être qu'un premier pas vers une réelle exploitation de tels environnements contraints. L'informatique opportuniste (Opportunistic Computing) dépasse le cadre des seules transmissions, et introduit un nouveau paradigme d'exécution de tâches collaboratives dans de tels environnements.

Dans ce domaine qu'est l'informatique opportuniste, la conception, la mise en œuvre et le déploiement d'applications distribuées sont des objectifs majeurs. Une application pour OppNet doit pouvoir fonctionner et assurer un niveau de service satisfaisant, tout en supportant les diverses contraintes propres aux OppNets, telles qu'une connectivité fluctuante, un partitionnement chronique du réseau, de longs délais de transmissions, de fréquents échecs de transmission, et des équipements hétérogènes offrant des ressources limitées.

La complexité et le coût du développement d'applications pour OppNets peuvent être réduits de manière significative en utilisant des modèles de programmation appropriés. De tels modèles peuvent être fournis par des systèmes intergiciels capables de supporter de manière transparente les contraintes évoquées plus haut.

Le travail rapporté dans ce mémoire a porté sur l'étude des contraintes inhérentes aux OppNets, et sur la proposition de solutions appropriées. Parmi les modèles de programmation usuels, certains ont été identifiés comme pouvant être utilisés dans le cadre des OppNets. Sur la base de ces divers modèles de programmation, des systèmes intergiciels opportunistes ont été mis en œuvre. Ces systèmes supportent respectivement le modèle de messagerie distribuée (sur la base de files d'attentes et de "topics"), le modèle du tuple-space, et la résolution de consensus. Des implémentations complètes ont été réalisées, et le code source est distribué sous licence GPL (GNU General Public License). Ces systèmes ont été évalués par le biais d'expérimentations menées en conditions réelles et par simulation.

Contents

List of Figures

List of Tables

Part I

Preliminaries

1
Introduction

Contents

MOBILE devices become part of our everyday life. These devices include laptops, smartphones, PDAs, sensors and even smartwatches. With the advances of technology, they are not anymore considered as electronic gadgets with limited functionality. Their usage is becoming a way of life for a huge number of users, since it fundamentally changes the way in which users organize their professional and private lives. Indeed, it is difficult to define in one way their usage, since daily activities are becoming more and more related to mobile devices: they are used for working, shopping, learning, playing, exchanging email, etc.

Using mobile devices, communication between users becomes easier and services become more available while on the move. Current wireless technologies (e.g., GSM or Wi-Fi) connect mobile devices using fixed infrastructures, with each devices are connected to wired networks via a base station or wireless access point "hotspot", etc. However this dependency on a fixed infrastructure from the one hand is expensive, not suitable for some situations (military and emergency situations), and from the other hand limits the flexibility of mobile devices.

The breakdown of one of these central points or simply trying to communicate off-base implies that the mobile devices will not be anymore able to intercommunicate even with adjacent mobile devices. As a result, some users will find themselves eventually isolated from each other even though they sit near each other.

For the sake of illustration, a simple scenario will be developed through this section. It aims for better emphasizing the importance of eliminating the dependency on centralized systems while connecting mobile devices. In this scenario, a group of friends are camping together in a site where no infrastructure exists. At night, the friends need to communicate with each other so as to share photos they have taken during the day using their smartphones. However, mobile devices are totally *isolated* in this area, where no Internet connection nor mobile phone coverage is yet available, as shown in Figure 1.1. Hence, the friends have to wait for any type of connection in order to exchange the photos they have taken while they were off-the-grid.

The idea of a network that reaches out to all mobile devices must hence be comple-

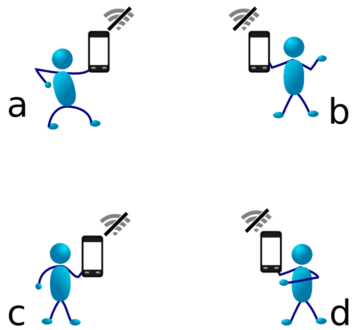


Figure 1.1: Isolated mobile devices in an area where no Internet connection is currently available

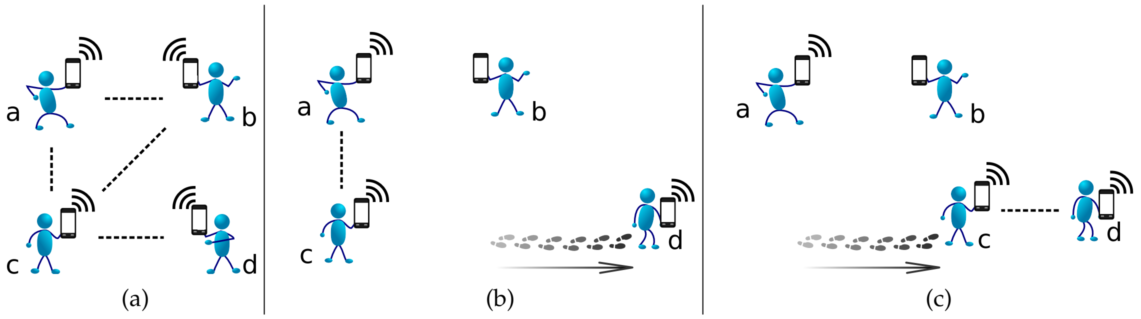


Figure 1.2: Example of an ad hoc network (dotted lines represent direct connections between mobile devices)

mented by another solution, by which mobile devices could interconnect directly any time and anywhere (i.e., in an *ad hoc* manner) to form their own network without the aid of a central infrastructure.

This vision of ad hoc networking led to the appearance of an emerging type of wireless networks, in which mobile devices associate on an ad hoc basis without any physical networking infrastructure. These wireless networks are usually mentioned in the literature as *Mobile Ad hoc Networks (MANETs)*. For our scenario, the friends can leverage ad hoc networking in order to form a MANET between themselves as shown in Figure 1.2a. By doing so, they can now exchange their photos.

MANETs play a pervasive role enabling users to form a network in many situations, such as the situation depicted in our scenario, where users cannot rely on an existing network infrastructure to do so: it may be unreliable, too expensive, inadequate, malfunctioning or simply unusable. In such situations, traditional networking approaches are considered as completely useless.

Mobile devices in a MANET can communicate *directly* with any other mobile device within transmission range. Furthermore, if two devices are out of range, they may still be able to relay on other devices in the MANET that forward data on behalf of each other until it reaches the final destination that would be considered otherwise as out of range. In the MANET shown in Figure 1.2a, imagine that user *a* wants to send some photos to user *d*. To do so, the mobile device of user *a* will forward these photos to some intermediate mobile devices, that are within reach and are situated closer to the final destination.

If the latter still cannot reach directly the mobile device of d, the photos will be further relayed to other intermediate devices, which are closer to the final destination. And so forth, mobile devices continue relaying photos till it reaches the mobile device of user d. Indeed, in MANETs a continuous end-to-end connection must be established before a message is exchanged, meaning that the sender and the receiver must stay simultaneously connected to a common network.

Unfortunately, in real conditions, due to user mobility, limited power supply of their mobile devices and the intrinsic wireless link instability, end-to-end connections between mobile devices might never exist. In our example, it could be possible that user b switches off his/her smartphone to preserve its battery budget and user d moves away from user c as shown in Figure 1.2b. By doing so, they prevent any end-to-end connection between a and d, and this of course prevents them from exchanging photos.

A MANET can therefore become disconnected when the mobile devices that compose this network are sparsely or irregularly distributed. The term *opportunistic network (OppNet)* is often used in the literature to denote a disconnected kind of MANET, where radio contacts between mobile hosts are not planned in advance and must thus be exploited opportunistically. OppNets represents the topic of this dissertation. In these networks, the mobility pattern of each host is actually that of its physical carrier. If the physical carriers in a disconnected MANET are vehicles or robots with pre-defined travel plans, then radio contacts between the hosts they carry can be predicted accurately, and routing strategies can be devised based on contact predictions. This is however not the case of OppNets, where the physical carriers are for example human beings carrying laptops or smartphones, or animals carrying sensors. In these networks radio contacts cannot always be predicted, although each contact is still an opportunity for two hosts to exchange messages.

In order to bridge the gap between non-connected parts of an OppNet, a main obstacle, i.e., mobility, is turned into an opportunity by allowing mobile devices to carry the required messages with them while moving, until they encounter another device deemed more suitable to bring the message (closer) to the eventual destination. In the literature, this paradigm is called the *store-carry-and-forward* mechanism. In our example, the mobile device of user a can for example forward the required photos to the mobile device of user c, which carries them while moving so as to forward them later to the mobile device of user d, as shown in Figure 1.2c.

Supporting networking in OppNets is only a first step. *Opportunistic computing* goes beyond the concept of opportunistic networking in OppNets and provides a new paradigm to enable collaborative computing tasks over these environments. Opportunistic computing is actually becoming a new distributed computing paradigm, involving transient and unplanned interactions between mobile devices. In this paradigm, all pervasive and available communication opportunities are exploited to provide computing services to meet application needs by leveraging available computing resources.

A wide variety of application domains would benefit from the opportunistic computing paradigm. However, designing and implementing distributed applications in the context of opportunistic computing is not a trivial task given the problems imposed by the different characteristics of OppNets. Indeed, OppNets are characterized by intermittent connectivity, long transmission delays, transmission failures, resource (power, memory and bandwidth) constraints, frequent changes in topology and heterogeneous devices. Some of the problems imposed by these characteristics are in common with MANETs, however, on a much larger scale and much more heighten.

Application development has been made easy and cost-effective in traditional, fully-connected networks by using middleware systems. Indeed, these middleware systems provide developers with a set of high-level programming models, which help them to easily develop their distributed applications. One can argue that this set of programming models could grant the same benefits for opportunistic computing.

However, existing middleware systems do not adequately address the many novel and fundamental challenges presented by OppNets, since they are not designed to tolerate unpredictable message loss or extremely long transmission delays. Hence, they cannot provide this set of high-level programming models in the context of OppNets.

All these reasons yield an increasing need for middleware systems specially designed for OppNets that, while coping with OppNets issues, provide the developers with this set of high-level programming models, that eases the opportunistic computing.

1.1 Objectives

Much of the complexity and cost of building OppNet-dedicated applications can be alleviated by the use of high-level programming models. Such models can be supported by middleware systems capable of transparently addressing all the problems that can occur in OppNets.

The work reported in this dissertation focused on providing insight into the fundamental problems posed by OppNets, so as to analyze and solve the problems faced by application developers while dealing with these environments. The research focused on identifying well-known high-level programming models that can be satisfactorily implemented for OppNets, and that can prove useful for application developers. In order to demonstrate the feasibility of application development for OppNets, while assessing the benefits brought about by carefully designed middleware systems, a couple of such systems have been designed, implemented, and evaluated in real settings and simulations as part of this work.

1.2 Contributions

The original contributions of this work are the following:

- Providing insight into the fundamental problems posed by OppNets, so as to analyze and solve the problems faced by application developers while dealing with these environments.
- Identifying well-known high-level programming models that can be satisfactorily implemented for OppNets, and that can prove useful for application developers.
- Demonstrating the feasibility of application development for OppNets, while as-

sessing the benefits brought about by carefully designed middleware systems, through the design of the following systems:

- An opportunistic, message-oriented middleware system to provide two important programming models in OppNets: the publish-subscribe programming model and the message queue programming model. This system, called *JOMS (Java Opportunistic Message Service)*, is actually a provider for the standard Java Message Service (JMS). Standard JMS applications using JMS message queues and topic can be directly deployed and executed in OppNets..
- A distributed directory service, that implements a subset of the standard Java Naming and Directory Interface (JNDI).
- A fully-distributed, peer-to-peer coordination middleware system to provide the tuple space programming model in the context of OppNets. This system, called *JION* (*JavaSpaces Implementation for Opportunistic Networks*), is actually a JavaSpaces implementation with support for Future object. Standard JavaSpaces applications can be directly deployed and executed in OppNets.
- A consensus middleware system to provide consensus services in OppNets. This system, called *ADAM* (*Agreement in Disconnected Adhoc Mobile networks*), is designed to enhance the performance of programming models that require some collaboratively-managed decisions.

• Supplementing the above-mentioned systems with fully-functional implementations, that can be used in real settings. These implementations are actually implementations of standard specifications so as to make them user-friendly. By doing so, developers do not need to learn a new programming language, or get familiar with an exotic programming model or API. Developers can simply focus on writing a standard application, and the middleware will take care of its execution in an OppNet.

• Realizing real-life experiments and simulations to evaluate the effectiveness and efficiency of the proposed systems. Indeed, real-life experiments on Linux and Android platforms have been realized in the context of this dissertation in order to evaluate the proposed middleware in real conditions.

• Distributing the proposed systems under the terms of the GNU General Public License (GPL) so as to be openly available for application developers.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 provides the necessary background about the different technologies, protocols and architectures used in this dissertation. The chapter first introduces OppNets and highlights the main issues around such challenged environments. It then gives an overview of the opportunistic computing paradigm, supplemented by several real-life scenarios. The chapter then provides an overview of the state-of-the-art in the field of high-level programming models, discussing the characteristics of each type of programming model. Afterward, the chapter discusses the main characteristics required in a programming model in order to be usable in the context of opportunistic computing. The chapter finishes by identifying a set of major types of high-level programming models, that is needed as a basis for opportunistic computing.

Chapter 3 conducts a literature study highlighting the communication middleware systems designed for OppNets. It starts by introducing the communication middleware systems that are openly-distributed to support communication in OppNets. Then, it illustrates the importance of content-centric networking in the context of OppNets.

Chapter 4 presents an opportunistic, message-oriented middleware system that provides two important programming models in OppNets: the publish-subscribe programming model and the message queue programming model. This system, called *JOMS*, is actually a provider for the standard Java Message Service (JMS). The chapter introduces JMS, describes its key features and explores how to make it disruption-tolerant so as to be able to run in OppNets. The design and implementation of JOMS are then discussed. Experimental results obtained during real-life measurement campaigns are finally presented and discussed.

Chapter 5 presents a fully-distributed, peer-to-peer coordination middleware system that provides the tuple space programming model in the context of OppNets. This system, called *JION*, is actually a JavaSpaces implementation with support for Future object. The chapter starts by giving some background about the basic concepts of JavaSpaces and Future objects, along with their importance in the context of OppNets. It then presents the design and implementation of JION. The chapter concludes by developing a real application using JION, which is then used to evaluate the efficacy of JION in real-life measurement campaigns.

Chapter 6 presents a consensus middleware system that provides consensus services in OppNets. This system, called *ADAM*, is designed in order to enhance the performance of programming models that require some collaboratively-managed decisions. The chapter starts by briefly reviewing the consensus problem in general. It then goes on to examine how a consensus can be solved in an OppNet. The chapter then discusses the design and implementation of ADAM. The chapter finishes by presenting and discussing the results obtained during experimental results and simulations.

Chapter 7 gives a conclusion summarizing the work presented in this dissertation, and outlines future works.

2

Background and Literature Review

Contents

THIS chapter provides the necessary background about the different technologies, protocols and architecture used in this dissertation. The targeted network platforms of the middleware systems designed in this dissertation are OppNets. This chapter introduces MANETs. An overview of OppNets and the main issues around these challenged environments represent the second point of this chapter. The chapter then gives an overview of the opportunistic computing paradigm in these networks, supplemented by several real-life scenarios. Consequently, an overview of the state-of-the-art in the field of high-level programming models is reviewed, discussing the strengths of each type of model. The motivation for designing middleware systems, that provide these programming models for OppNets is finally presented.

2.1 Introduction to MANETs

Mobile ad hoc networks (MANETs) have justified a fair amount of research activity during the last two decades. A MANET, unlike traditional communications networks, is both self-forming and self-healing, enabling peer-level communication between mobile devices (so-called *nodes*) without reliance on centralized resources or fixed infrastructure. Connectivity is supported by communication technologies allowing for direct communications between nodes, of which Bluetooth and Wi-Fi are the most popular solutions. Communication channels are established when an encounter occurs between two nodes. That is, when they are within radio range. The range of communication technologies varies with the frequency band, radio power output, antenna gain, antenna type and indoor/outdoor situations. Normally, the range of Bluetooth is about 5/30 meters (indoor/outdoor), whereas the range of Wi-Fi is about 35/120 meters (indoor/outdoor).

By eliminating the reliance on any pre-existing fixed communication infrastructure, MANETs offer rapid and easy network deployment in many situations, previously not possible, where the existing infrastructure is unreliable, too expensive, inadequate malfunctioning or simply unusable.

The idea of being able to form a network in ad hoc mode, send/receive messages in an infrastructureless environment has a number of applications, from providing state-of-the-art military and emergency services where existing infrastructure is unreliable or inaccessible, to an easy and efficient communication in meeting conferences where planning and setting up an infrastructure would be too time consuming. More details about other applications that can benefit from these networks can be found in [1].

Routing in MANETs has attracted much attention from the research community, since it allows nodes to reach beyond their radio coverage, sending messages to non-directly connected nodes. The many routing protocols designed for fixed networks are ineffective in MANETs, which completely disregard their basic assumption: nodes in MANETs cannot be considered as continuously available, reachable via a known route or having a fixed relationship with other nodes. Indeed, an optimal route at a given time might not work a few moments later.

To this end, dozens of multi-hop forwarding protocols have been put forward for use in MANETs (such as OLSR, AODV, DYMO, DSR...), allowing nodes to rely on each other to send messages, forming a multi-hop wireless network (a good survey can notably be found in [2]). The notion of *reachable node* is a key feature in MANETs routing protocols. It is defined as follows: a node is considered as *reachable* from another node, if there actually exists at least one temporaneous end-to-end path between them.

Based on how routing information is acquired and maintained by mobile devices, MANETs routing protocols may be classified into three main categories: proactive routing, reactive routing and hybrid routing.

In proactive routing protocols, nodes continuously evaluate routes to all other *reachable* nodes in a MANET. By doing so, every node attempts to keep its routing table updated regardless of the actual communication needs of the node. The primary advantage of proactive protocols is the fast connection setup: when a node wants to send a message to a destination node, there is a high probability that a routing path to the destination node does already exist in the routing table. However, the proactive nature of this type of routing protocols involves a non-negligible overhead for constantly maintaining up-to-date routing information.

In reactive routing protocols, routes are determined solely when they are explicitly needed. That is, when a node needs to send a message to a destination node, a route-discovery procedure is first invoked. This procedure terminates either when a route to the destination has been found and the node is hence considered as a *reachable* node, or simply no route is available for the *non-reachable* node. The obvious advantage of reactive protocols is that they limit route maintenance required by proactive protocols. Hence, they require less overhead than proactive protocols, making them more scalable in MANETs. Their main disadvantage is the long delays incurred during route discovering before every messages can be forwarded.

Hybrid routing protocols combine the merits of both proactive and reactive routing protocols to overcome their shortcomings. Hybrid protocols group nodes into clusters of nodes. Basing on this hierarchical network architecture, proper proactive protocols and reactive protocols are exploited at different hierarchical levels, enabling message forwarding in MANETs.

The main conclusion drawn from the aforementioned overview is that messages can only be forwarded when the destination node is reachable.

2.2 Targeted Network Platform

The targeted network platforms in this dissertation are OppNets. This section introduces these networks. It then discusses the challenges, that should be kept in mind while dealing with OppNets.

2.2.1 From MANETs to OppNets

The notion of reachable node is a key feature in MANETs routing protocols. Unfortunately, this assumption cannot always be held on; as nodes are sparse in a system, encounters that depend on the mobility of nodes become sparse. As a result, many real MANETs are, under the most favorable conditions, only partially or intermittently connected.

The sparse or irregular distribution of mobile devices in a MANET can, for example, induce link disruptions in this network. These disruptions may in turn split the network into a collection of distinct, continuously changing, disconnected "islands" (connected components) as shown in Figure 2.1. Within every island, communication between hosts is possible –using multi-hop forwarding if needed– but no instant communication is possible between hosts that reside on different islands. As a consequence, hosts in a given island will find themselves isolated from hosts in other islands. In such conditions, a method must be devised in order to bridge the gap between non-connected parts of the network.

If the physical carriers in a disconnected MANET are vehicles or robots with predefined travel plans, then radio contacts between the hosts they carry can be predicted accurately, and routing strategies can be devised based on contact predictions. But if the physical carriers are for example human beings carrying laptops or smartphones, or animals carrying sensors, and the mobility pattern of each host is actually that of its physical carrier; then radio contacts cannot always be predicted, although each contact is still an opportunity for two hosts to exchange messages. The term *opportunistic networks (OppNets)* is often used in the literature to denote a disconnected MANET, where radio contacts between mobile hosts are not planned in advance and must thus be exploited opportunistically. OppNets represents the context of this dissertation.

The "*store, carry and forward*" approach, the foundation of Delay/Disruption-Tolerant Networking (DTN) [3], is usually proposed for solving intermittently connected networking problems, which is the case in OppNets. Hence, it can help bridge the gap between non-connected parts of an OppNet: the mobility of hosts makes it possible for messages to propagate network-wide using mobile hosts as carriers (or *data mules*), that can move between network islands. As shown in Figure 2.1, connectivity disruptions between islands 1 and 2 can for example be tolerated thanks to users moving (deliberately or by chance) between these islands. The device of a user moving from island 1 to island

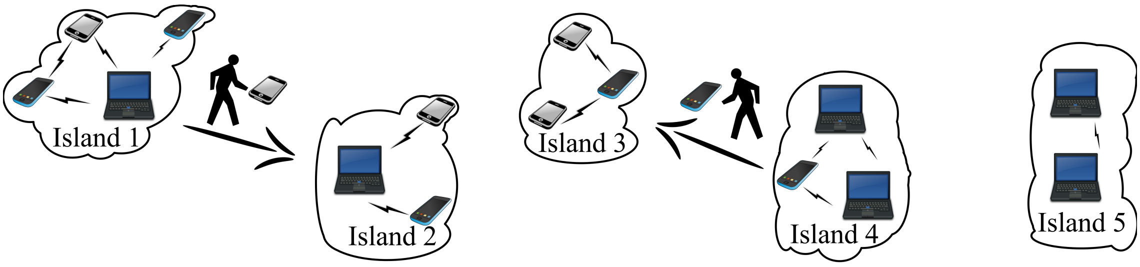


Figure 2.1: Example of an opportunistic network composed of user-carried mobile devices

2 acts as a data mule for messages addressed to hosts located in island 2. Considering that many carriers may be involved successively for the transmission of a single message, this approach provides message delivery at the price of additional transmission delays.

In any case, it should be noticed that communication protocols for OppNets can usually provide no more than best-effort delivery: they cannot guarantee that a message will be delivered. This is a consequence of the disconnected nature of the networks considered. Consider again the example shown in Figure 2.1, and assume a message is addressed by a host in island 1 to a host in island 5 (or to all hosts in the network, including those in island 5). If no human carrier ever visits island 5, then there is no chance that the message ever gets delivered in this island. A communication protocol running in an OppNet can do no magic if mobile hosts do not move in such a way that messages can be transported between non-connected fragments of the network.

2.2.2 System Model

This section presents the system model of the OppNets considered in this dissertation.

General architecture The system model consists of a finite set of mobile nodes $\mathcal{V}$. Each node features a short-range wireless interface, that allows it to exchange messages in ad hoc mode –that is, without relying on any fixed infrastructure– with nodes in its radio range.

At any time a node is either up or down: a node can disable its wireless transceiver or enter standby mode spontaneously to save battery, or it can be switched off and on alternatively by a user. This behavior is normal and is not considered as a failure. When a node is down, it cannot communicate with its neighbors. When it is switched on again, its previous state is restored, and it initiates a neighbor discovery phase in order to adjust rapidly to its current surroundings.

No assumption is made about the mobility of nodes, or about their spatial distribution. A node can thus be sometimes isolated from the other nodes, when the distance to its closest neighbor exceeds its radio range.

The relations between nodes take place over a time span $\mathcal{T}$. At some time in $\mathcal{T}$ the system model is represented by the static graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of nodes and $\mathcal{E}$ the set of edges. There exists an edge between two nodes u and v if u

and v are within mutual radio range, and can thus exchange messages over the wireless medium. Over time, the system model can be represented by a *time-varying graph* (TVG) $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \rho, \psi)$, where $\rho : \mathcal{E} \times \mathcal{T} \rightarrow \{0, 1\}$ is the edge presence function, that indicates whether a given edge is available at a given time; and $\psi : \mathcal{V} \times \mathcal{T} \rightarrow \{0, 1\}$ is the node presence function, that indicates whether a given node is available at a given time [4]. At some time in $\mathcal{T}$, the underlying graph G of $\mathcal{G}$ is not necessarily connected. In fact, the underlying graphs of $\mathcal{G}$ may all be disconnected over $\mathcal{T}$.

Store-carry-and-forward model In the TVG formalism, a *journey* in $\mathcal{G}$ is defined by a sequence of couples $\{(e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n)\}$, such that $\{e_1, e_2, \ldots e_n\}$ is a walk in G and $\rho(e_i, t_i) = 1$ and $t_{i+1} > t_i$ for all $i < n$ [4]. At time t, a message sent by a node u may eventually be received by a node v if a *journey* $u \rightsquigarrow v$ exists in the TVG $\mathcal{G}$, with a starting date t_1 such that $t_1 \geq t$. In other words, a message sent by u may reach v after being stored, carried, and forwarded successively by several nodes, even if no end-to-end path between u and v never exists in any underlying graph G of $\mathcal{G}$. The time elapsed between two consecutive hops $(t_{i+1} - t_i)$ may range between a few milliseconds (the message being forwarded rapidly between successive neighbor nodes) and minutes or hours (the message being carried for a while before being forwarded to another node).

Fault model The nodes in this system model can be switched off and on at any time. When a node is switched off this is not considered as a “failure”, for this is consistent with its normal behavior. Nodes may sometimes crash spontaneously and never recover. Using the TVG formalism, the notation $\psi(v, t) = 0$ is used to indicate that the node v is switched off at time t.

In an OppNet, any wireless link between two neighbor nodes is inherently transient. However, journeys between any pair of nodes do not require any temporaneous end-to-end connectivity between these nodes, and can thus rely on successions of transient links. As a consequence, disruptions of wireless links are not considered as “link failures” in this model.

The communication model cannot guarantee that each message sent in the network eventually gets delivered to all possible recipients. This system model therefore admits *receive omissions*.

2.2.3 The Cornerstone of OppNets

In the wide variety of work published over the past years, researchers identified various types of challenged networks (e.g., MANETs, VANETs, D-MANETs, ICNs, DTNs, etc.) and thoroughly described the characteristics of these networks and the requirements to support networking in them. Each network type was illustrated by several real-life scenarios [5]. However, there is not a common agreed terminology and a clear separation of concepts for these challenged networks in the literature. Indeed, the different terms of these networks are used interchangeably to refer to a specific network. A typical example is the use of the terms “challenged networks”, “opportunistic networks” and “delay/disruption-tolerant networks” in an interchangeable manner. According to Pelusi et al., *the terms “opportunistic networks” and “delay-tolerant networks” are often used* *interchangeably. In our view, [...] opportunistic networks correspond to a more general concept and include DTNs* [6].

Hence, before starting dealing with OppNets, it is worthwhile in this regard to concretely define these concepts so as to:

1. identify the different types of challenged networks.

2. identify the main characteristics of OppNets that make them qualify as challenged networks.

2.2.3.1 Challenged Networks

Challenged networks are basically all *networks* that do not meet Internet design assumptions: the existence of an end-to-end path(s) between any arbitrary source-destination pair or a bounded round trip time between source and destination [7]. According to Kevin Fall, challenged networks *are qualitatively characterized by latency, bandwidth limitations, error probability, node longevity, or path stability that are substantially worse than is typical of today's Internet. [...] The use of the Internet's performance as a baseline is due to its enormous scale and influence* [7].

Delay/Disruption-Tolerant Networks (DTNs) are *challenged networks* characterized by very long transmission delays resulting from either:

1. physical link properties in networks where messages must be transferred over vast distances (e.g., between planets). Networks characterized by very long propagation delays are usually referred to as Delay-Tolerant Networks. Interplanetary Internet (IPN) is an example of this type of networks [8].

2. extended periods of network partitioning resulting from frequent disconnections, making continuous end-to-end connectivity difficult or impossible to be guaranteed. Networks characterized by frequent disconnections are usually referred to as Disruption-Tolerant Networks. An example of this type of networks are Tactical Networks [9].

The Delay-Tolerant Networking Research Group (DTNRG)[1] is a research group chartered as part of the Internet Research Task Force (IRTF) to address the architectural and protocol design principles for Delay/Disruption Tolerant Networking, arising from the need to provide communications in Delay/Disruption-Tolerant Networks [10]. DTNRG acts as a central point where protocol specifications in the context of DTNs can be agreed and standardized.

The term Delay/Disruption-Tolerant Networks (DTNs) is sometimes used as a reference to Delay/Disruption Tolerant Networks, at other times it is used as a reference to Delay/Disruption Tolerant Networking. In the remainder of this dissertation, we shall use the term Delay/Disruption-Tolerant Network (DTN) to reference the network (by itself), which is exploited using Delay/Disruption-Tolerant Networking (DTNing). Along the

[1] `http://www.dtnrg.org`

same lines, the term DTNing architecture shall be used to refer to the Delay-Tolerant Networking Architecture proposed by the DTNRG. This naming convention is used along this dissertation.

According to Cerf et al., the DTNing architecture *provides a common method for interconnecting heterogeneous gateways or proxies that employ store-and-forward message routing to overcome communication disruptions* [10]. To this end, a DTN, built using the concepts of DTNing architecture, is based on the store-and-forward technique, which implies that the participating devices are not necessarily *mobile*.

Mobile Ad hoc Networks (MANETs) are challenged networks characterized by node mobility and ad hoc connectivity between nodes. The main assumption in MANETs is such that a network-wide end-to-end connectivity is always guaranteed is these networks despite the problems posed by their main characteristics.

Opportunistic Networks (OppNets) represent another type of challenged networks where problems occur frequently due to unpredictable node mobility, ad hoc connectivity, sparse density and unreliable links. In OppNets, unlike in MANETs, it is impossible to guarantee an end-to-end connectivity between different nodes. That is, OppNets are challenged networks that combine the characteristics of DTN and MANETs: OppNets are characterized by very long transmission delays, unpredictable node mobility and ad hoc connectivity. Furthermore, OppNets are characterized by opportunistic connectivity between mobile nodes. OppNets are referred to in the literature under different terms, such as challenged networks, intermittently connected networks (ICNs) [11], Pocket Switched Networks (PSNs) [12] or Disconnected MANETs (D-MANETs) [13]. The main reason behind the existence of these different terms is that the researches in this domain were all started at the same time. Yet, these terms refer to similar types of networks in which:

1. the most of the nodes (or maybe all of them) are mobile.
2. mobility patterns are not known in advance (i.e., not predictable) and contacts between nodes must be exploited opportunistically.
3. nodes get connected together in an ad hoc manner.
4. it is not possible to guarantee an end-to-end path between any pair of nodes. As a result, transmission delays are extremely long.

To summarize, DTNs are challenged networks characterized by very long transmission delays. MANETs are challenged networks characterized by nodes mobility and ad hoc connectivity between nodes. OppNets are challenged networks characterized by very long transmission delays, unpredictable nodes mobility and opportunistic connectivity between nodes.

2.2.3.2 OppNets as Challenged Networks

According to Thrasyvoulos et al., three main dimensions must be considered when dealing with challenged networks, namely *connectivity*, *mobility* and *network and node resources* [14]. This section is dedicated to give a closer look on the main characteristics of OppNets.

Connectivity OppNets are considered as disconnected challenged networks, in which nodes are isolated most of the time, or have at most a few neighbors. Every now and then, two nodes come into contact for a limited duration, during which they can exchange useful information.

Disconnections in OppNets are either *involuntary disconnections* or *deliberate disconnections*. An involuntary disconnection is usually forced by external factors (e.g., nodes mobility or external interferences). Involuntary disconnections occur frequently and unpredictably in real conditions. A deliberate disconnection is a disconnection enforced by users to save cost or energy. This kind of disconnections could be sometimes predictable.

In OppNets, connectivity disruptions are predominantly tolerated using the store-carry-and-forward mechanism along with the concept of opportunistic forwarding. Indeed, every radio contact between nodes is exploited opportunistically since it represents a valuable opportunity for them to exchange messages. Usually this comes at the price of extremely long transmission delays, and transmissions can even fail altogether.

Mobility Mobility patterns may provide the ability to predict future contacts in a given challenged network. Hence, it deserves a central place in the study of challenged networks.

The mobility patterns of nodes that compose the network considered can be assumed to be known in advance, so contacts between these nodes can be predicted accurately. This is typically the case when the carriers are buses or garbage trucks with well-defined travel plans. It can also be assumed that the carriers are robots whose mobility patterns can be controlled as needed in order to satisfy communication needs. Furthermore, it can be assumed that the carriers are human beings (as illustrated in Figure 2.1), whose mobility patterns can be predicted by identifying the way by which these people move or meet, or by identifying what communities each person belongs to.

In OppNets, no such assumption is made about the carriers of mobile nodes and propose to rely on redundancy in order to improve the reliability of delay-tolerant transmission. Indeed, carriers could be human beings, but they could as well be vehicles, robots, animals, or any combination of these. In these environments more or less controlled forms of epidemic or probabilistic propagation schemes are used to ensure that each message is duplicated and transported by several carriers.

Network and Node Resources

Network Resources OppNets are faced with the traditional problems inherent to wireless communications. Wireless links have time varying characteristics in terms of link capacity and link error probability. The fact that the surrounding environment interacts with the wireless signal makes the wireless communications less reliable and provide significantly lower capacity than wireline communications.

In addition, the realized throughput of a wireless link –after accounting for the effects of multiple access, fading, interference conditions, etc.– is often much less than its theoretical maximum transmission rate.

Node Resources Depending on the kind of nodes that compose the network considered, the resources of nodes such as storage and battery may vary largely. Although the resources of vehicular nodes are not considered as an issue in a VANET, it is not typically the case for simple mobile devices carried by pedestrians in other kinds of challenged networks. Resources may also be different among nodes in some kinds of challenged networks, in which nodes with significant resources may coexist with small resource-constrained nodes. This is for example the case in hybrid networks, in which infrastructure-less ad hoc networks are mixed with infrastructure-based networks so as to get benefit from the reliability and robustness of infrastructure-based networks [15, 16]. The availability (or the lack) of node resources in a given network affects the way by which the network should be dealt with.

An important factor in OppNets is the resource-constrained nature of mobile devices, which are generally characterized by limited processing power, memory size, and power budget. Mobile devices are generally light and small to be easily carried around, and hence they have also size and weight constraints. Such considerations in conjunction with a given cost and level of technology, will keep mobile devices having less resources than personal computers.

2.3 Opportunistic Computing

OppNets constitute an appealing solution to complement fixed network infrastructures –or make up for the lack thereof– in challenged areas. Supporting networking in OppNets is only a first step. *Opportunistic computing* goes beyond the concept of opportunistic networking in OppNets and provides a new paradigm to enable collaborative computing tasks over these environments.

According to Conti et al., *when two devices come into contact, albeit opportunistically, it is also a great opportunity to share and exploit each other's (software and hardware) resources, exchange information, cyber forage, and execute tasks remotely. This opens a new computing era: the opportunistic computing era [17].*

Opportunistic computing is actually becoming a new distributed computing paradigm, involving transient and unplanned interactions between mobile devices. Opportunistic computing generalizes the concept of opportunistic networking, considering the possibility for mobile devices to opportunistically leverage each others' general resources, including, but not limiting to, heterogeneous hardware components, software processes, multimedia content, sensors and sensory data. While not all resources can be available on any single device, they can be collectively available to provide computing services to meet application needs through the paradigm of opportunistic computing.

The opportunistic computing paradigm enables several interesting applications that are already being investigated in the areas of crisis management, wildlife monitoring, challenged environments and other different areas as discussed in the next section.

2.3.1 Applications Areas

Despite the fact that few applications have been so far implemented for the context of opportunistic computing, many potential applications will probably follow in the next few years knowing that their real case scenarios already exist. These scenarios include:

Disaster Scenarios Disaster scenarios such as fire, flood or earthquake are amongst the earliest scenarios which have been devised for opportunistic computing. Indeed, disaster relief efforts normally take place where network infrastructures are destroyed or simply become inaccessible. Taking into consideration the short range of wireless transmission of mobile devices and the size of the area affected by the disaster, the absence of end-to-end connectivity between mobile devices is not considered as an exceptional event. Obviously, in such scenarios, opportunistic computing is a necessity where only this paradigm can survive and provide an efficient means to support collaborative computing services to applications and users.

Inter-Vehicular Networks Vehicular Networks (a.k.a, Vehicular Ad hoc Networks or VANETs) emerged as a means to enable vehicles to communicate with each other as well as with roadside base stations to enhance traffic safety and reduce vehicles collisions. VANETs can support many other applications such as informing drivers about weather conditions or finding a place to park [18]. Until now, most of researches in VANETs are done under the over-simplified assumption that a VANET is a well-connected network in nature. To this end, VANETs are always considered as MANETs [19]. However, under many real scenarios (e.g., at nights when the number of vehicles diminishes noticeably) VANETs tend to be disconnected, consisting of a collection of disjoint, continuously changing islands [20, 21]. Rubinstein et al., have performed a number of experiments in real VANETs presented in [22]. Based on the obtained results, the authors motivate the use of OppNets techniques in VANETs. The research area of disconnected vehicular networks can leverage the opportunistic computing paradigm, which provides a basis for the development of opportunistic applications in these networks.

Ubiquitous Social Networking Recent technological innovations have brought to everyone the possibility of carrying smartphones. As a consequence, mobile social networking starts appearing as a trend that is recently occupying a large portion of an individual's daily life. Nowadays, almost everyone has a kind of social account (e.g., Facebook, Google+ or Twitter) through which people of common interests aim to interact with each other and exchange information on so many different levels. Ad hoc communication in such scenarios has received particular attention from the networking research community [12]. Indeed, the support of ad hoc communication in well-organized groups has emerged as an important MANET research area. Groups are well-organized in the sense that users proceed in group trying not to go far away from each other, e.g., soldiers in a battlefield. In contrast, OppNets are originally conceived for groups that are partitioned due to geographical separation or user movement. In these groups, opportunistic computing allows mobile devices (and applications executing on them) to share each other's content, resources, and services.

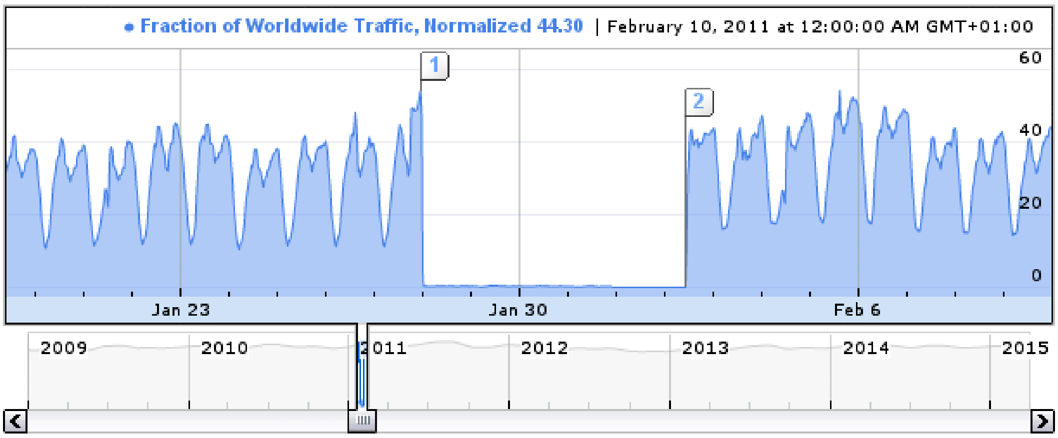


Figure 2.2: Google transparency report presenting an Internet outage in Egypt during the Arabic spring

Taking for example the *Arab Spring* that began on 2010 and spread throughout the countries of the Arab League, people using the power of communication were able to protest against their governments and show their dissatisfaction. Of course, authorities tried desperately to abridge communication channels by disturbing or even completely cut the Internet access. For example, in January 2011, the Egyptian authorities had succeeded in shutting down the country's international Internet access points for one week in a response to the public uprising. Figure 2.2 shows a Google transparency report which presents an Internet blackout in Egypt for one week. During this period, Egyptian people were neither able to intercommunicate nor to get their "tweets" and status updates out to the world. Indeed, they were in desperate need of an infrastructure-less model of communication to be able to intercommunicate. Such a real scenario represents one of the areas targeted by opportunistic computing, which proves itself as an especially resilient tool with which there is no easy way for an authority to prevent users from sharing essential resources and information.

Along the same lines, tens of thousands of protesters have taken over central areas of Hong Kong's financial district in September 2014 to protest against Beijing's decision to restrict Hong Kong election policy so as to let a mainland Chinese committee vet the city's political candidates. The protesters have worried that the Chinese government would block local phone networks. In response, they have used their phones to established an OppNet.

Opportunistic computing is not just for political upheavals, it could be used to provide collaborative computing services to a group of people doing fieldwork on several spots scattered over vast areas (e.g., geologists, naturalists, archaeologists, etc.). Each one in the group has a mobile device to communicate among themselves. However, the spots may be separated from each other by a distance, that is several times the radio range of the mobile devices. In such a situation, opportunistic computing can provide computing services to meet the pervasive application requirements.

Wildlife Monitoring Wildlife monitoring has received a lot of attention on both the biology and computer networking levels. It focuses on gathering data and observations on wild species range out in the wilderness with a goal of deeply investigating their behavior and understanding their interactions and influences on each other. In the wilderness, the strong need of biologists is to limit human intervention in order to respect the natural ecosystem (i.e., it is not advisable to provide a more structured network in such environments). Projects in this domain are ZebraNet for zebra tracking [23], SWIM for whale tracking [24] and TurtleNet for turtle tracking[2]. In these projects, animals under study carry sensors, that collect data which is then transferred to a destination processing center. Building a connected MANET between these sensors is quite impossible: as animals move, the sensors become spread throughout vast areas making it impossible to establish a connected MANET between them. Only OppNets can be established in this scenario, and opportunistic computing is used as a reliable, cost-effective and non-intrusive means to monitor these large populations roaming in vast areas.

Challenged Environments Providing intermittent Internet access to challenged environments (such as rural and developing areas) is another scenario in which opportunistic computing leads to accessing essential resources and information. Challenged environments usually lack communication infrastructure and reliable energy supplies. Deploying traditional (wired or wireless) networks to cover these areas is not cost-effective. OppNets are an affordable solution in order to bridge the digital divide.

Project DakNet [25] represents an example in this domain. It is aimed to provide low-cost digital communication to rural villages in India. A few connection-enabled computers (a.k.a., access points) are installed in these villages in order to provide villagers with some services (e.g., email, government services, etc.). Messages are all stored at the access points and are opportunistically forwarded to any connection-enabled vehicle passing by (i.e., buses, cars or bikes equipped with Wi-Fi) acting as mules. In turn, these mules carry the villagers' messages to the nearest city where they are forwarded over the Internet. Bytewalla [26] is a similar project, which aims to provide Internet access in remote villages in Africa.

The previously-mentioned scenarios represent a small part of overall scenarios that can benefit from opportunistic computing. Additional examples can be found in [27].

2.3.2 Challenges of Application Development in OppNets

In the realm of opportunistic computing, properly designing, implementing and deploying real applications are important tasks. Yet an OppNet-dedicated application must be able to operate and maintain an acceptable level of service while addressing the various problems imposed by these networks. In the OppNet literature, an OppNet-dedicated application is usually referred to as a *Disruption tolerant* application.

According to Sterbenz et al., disruption tolerance is *the ability of a system to tolerate disruptions in connectivity among its components, consisting of the environmental challenges: weak and episodic channel connectivity, mobility, unpredictably-long delay, as well as tolerance of energy (or power) challenges* [28].

Power budget represents one of the three essential and limited resources that form the

[2] `https://dome.cs.umass.edu/turtlenet`

major constraints for mobile devices: processing power, memory size along with power budget.

OppNet-dedicated systems, regardless of the specific implementation details and choices of different systems, must provide the following type of tolerances:

Error-prone Wireless Communications Tolerance this requires an application developed for OppNets to be able to deal with time varying throughput between nodes combined with the unreliability of wireless communication channels.

Mobility Tolerance link disconnections and network partitioning in OppNets are considered as the severe effects of node mobility in these networks. Because of the frequent disconnections and network partitioning in an OppNet, no host can be considered as accessible enough to play the role of a server for all the other hosts. Consequently, applications developers should generally use a peer-to-peer model rather than a client-server one. Furthermore, the frequent disconnections along with network partitioning make long transmission delays inevitable in OppNets. Hence, developers should opt for an asynchronous model while writing their applications. They should also consider possible transmission failures and design their applications properly to tolerate such failures.

Resource-constraint Tolerance the resource-constrained nature of mobile device in OppNets requires developers to establish the right trade-off between computational load and non-functional requirements (i.e., performance characteristics) achieved by their applications.

2.4 High-level Programming Models and Middleware Systems

According to Conti et al, *opportunistic networking, autonomic communication and computing, social networking, and multimodal sensing are enablers for opportunistic networking; however, realizing the opportunistic networking concept requires tackling several new research challenges for developing a set of middleware services that cope with disconnections and heterogeneities, and provide the applications with uniform access to data and services in a disconnected environment [17].*

In the light of the aforementioned statement, much of the complexity and cost of building OppNet-dedicated applications can be alleviated by the use of high-lever programming models provided through middleware systems aimed at transparently addressing all the main phases involved in applications developments in OppNets.

This section defines middleware systems. Then, it briefly reviews the major types of programming models available today, their main characteristics and the best applications for each kind in order to determine which programming models are well-suited to OppNets.

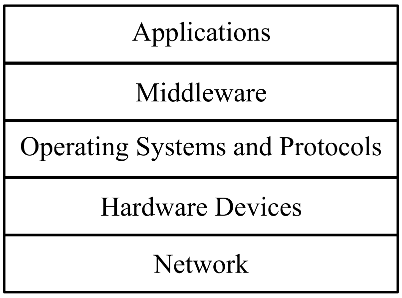


Figure 2.3: Layers of a middleware-based architecture

2.4.1 What is Middleware?

Since the 1980s, where the concept of middleware first came up, the term middleware has acquired several meanings, that would allow it to be just about any piece of software that sits between systems.

Nowadays, there are many categories of middleware systems that provide a broad set of programming models supporting a broad spectrum of distributed environments. Their main aim is to help developers to build their applications while being shielded from dependencies on underlying platforms (e.g., communication protocols, operating systems and hardware).

Till now, there is no single agreed-upon definition of middleware. In this dissertation, we adopt the middleware definition as presented in [29]:

The term middleware applies to *a software layer that provides a higher-level programming abstraction for the development of distributed systems and, through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability.*

Hence, a middleware-based architecture can be decomposed into multiple layers, which are shown in Fig 2.3.

2.4.2 High-level Programming Models: A Taxonomy

According to Coulouris et al., developers should consider two questions, which are central to an understanding of distributed systems [29]:

1. What are the entities that are communicating in the distributed system?
2. How do they communicate, or more specifically, what communication paradigm is used?

Addressing these questions together plays a major role in defining a reach design space for distributed systems. By doing so, the authors presented a categorization of middleware. This categorization follows five high-level programming models: distributed objects, distributed components, publish-subscribe systems, message queues, and Web services.

This categorization, according to the authors, is not exact and some modern middleware systems tend to support more complex programming models (that is, to offer hybrid solutions). For example, many distributed object platforms offer distributed event services to complement the more traditional support for remote method invocation. Similarly, many component-based platforms (and indeed other categories of platforms) also support Web service interfaces and standards, for reasons of interoperability. However, this categorization is intended to be indicative of the set of programming models required to build distributed applications in common distributed computing environments. The authors insist on the fact that this general taxonomy could be complemented by other models, such as distributed shared memory, tuple spaces or group communication.

According to Roman et al., *space and coordination are the two most critical dimensions to be considered in any systematic treatment of mobility* [30]. Along the same lines, researchers assert the importance of providing coordination models, such as tuple spaces, in mobile environments [31, 32]. To this end, in order to make that set of programming models well-suited for OppNets, it should be complemented by a tuple space programming model so as to provide coordination services in these environments.

This section highlights the relative strengths of each category of programming model and identifies their key characteristics, that divide them into major categories. More details about these programming models are given in the following chapters.

Distributed objects Distributed objects were among the earliest instances of programming models, which have become later a major area of study [31]. Distributed objects are designed to provide access to remote objects transparently, by extending the Remote Procedure Call (RPC) mechanism with the *object-orientation* concept. By doing so, it brings the benefits of the object-oriented approach to distributed programming: encapsulation, data abstraction, inheritance and polymorphism.

A plethora of middleware systems based on distributed objects are available (e.g., OMG CORBA [33] and Oracle Java/RMI [34]).

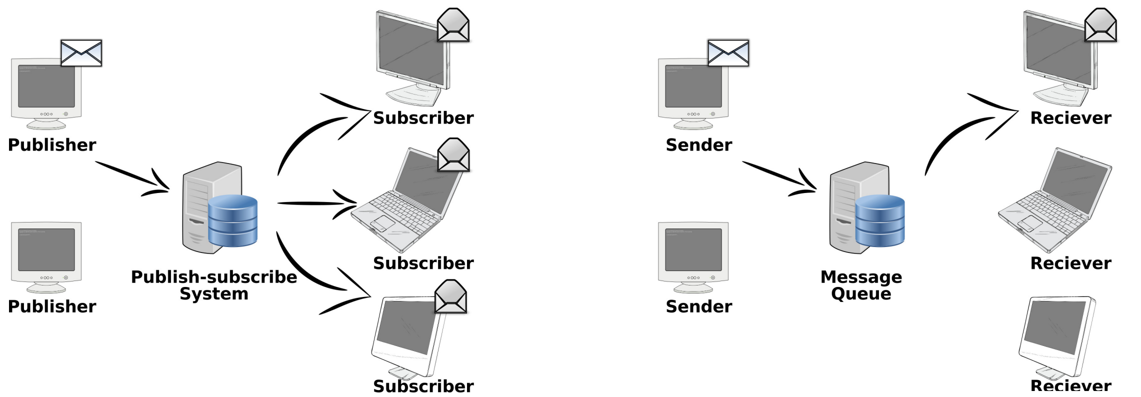
Distributed components Distributed components have emerged as successors to distributed objects programming model. They are considered as a direct response to address the weaknesses observed by application developers using the distributed object model (i.e., implicit dependencies, programming complexity, lack of separation of distribution concerns, and no support for deployment). More details about these problems can be found in [35].

According to Szyperski, a software component is *a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties* [35].

Distributed components can be best understood as a natural evolution of distributed objects, offering *problem-oriented abstractions* for building distributed systems.

Enterprise Java Beans, [36], Fractal [37] and CORBA Component Model [38] are well-know examples of middleware systems based on distributed components.

Publish-subscribe systems Many systems can be classified as information dissemination systems, wherein data need to be disseminated to a large number of clients. It would be complicated and inefficient to employ any of the traditional request/reply systems for



(a) Publish-subscribe system

(b) Message Queues

Figure 2.4: Communication model in publish-subscribe systems and message queues systems

this purpose. Hence publish-subscribe programming model has emerged to meet this important need [39]. Publish-subscribe model supports asynchronous, many-to-many communication systems in distributed systems, in which there are two different types of nodes: publishers and subscribers, as shown in Figure 2.4a. In these systems, several publishers disseminate information items (or messages) of interest to several subscribers. All communications between them are handled by a publish-subscribe system, removing dependencies between the publishers and the subscribers. That is, publishers and subscribers are *loosely-coupled*.

Some examples of middleware systems that provide the publish-subscribe programming model are: CORBA Event Service [40] and Java Message Service (JMS) [41].

Message queues While the publish-subscribe programming model offers a one-to-many style of communication, the message queues programming model offers a one-to-one communication model. This type of programming models is based on the concept of *queue* between a sender and a receiver in a distributed system, as shown in Figure 2.4b. A sender sends a message to a receiver through a central queue. It then becomes the responsibility of the queue to deliver it to the receiver. This indirect type of communication allows the sender and the receiver to be *loosely-coupled*: they do not have to be running at the same time in order to communicate.

JMS [41] and WebSphere MQ [42] are examples of middleware systems that support message queues.

Web services Web services represent an important paradigm of programming for the development of distributed applications, as they have been endorsed by many major software companies. This model simplifies interoperability between applications over networks [43]. It provides a means for wrapping existing applications so developers can access them through standard languages and protocols, such as XML and HTTP. Instead of interacting with heterogeneous systems, each with its own transport protocol, data format and interaction protocol, applications can interact with systems that are more

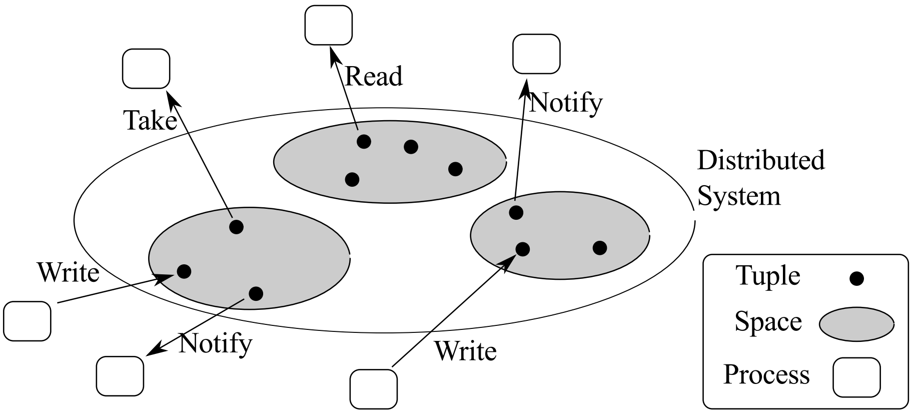


Figure 2.5: Tuple space architecture

homogeneous. As the result, developers can use the Web services programming model in order to build powerful information systems. Apache Axis2 [44] is a middleware system that provides this programming model.

Tuple spaces The tuple space programming model has evolved from the need to support coordination and distributed exchange of generic data structures in distributed systems, so as to allow communication partners to operate more efficiently. Indeed, the tuple space programming model allows participants (a.k.a., processes) to exchange generic data structures, called tuples, using a form of distributed shared memory [45], as shown in Figure 2.5. JavaSpaces [46] is a middleware system that provides this programming model.

For the sake of clarity, Table 2.1 summarizes the proposed categorization, providing some famous middleware systems for each programming model.

2.4.3 Providing High-level Programming Models in OppNets

Several middleware systems have been designed and are successfully used to provide the aforementioned high-level programming models in traditional distributed systems built on fixed networks. In the context of challenged ad hoc networks, different middleware systems have also been proposed to provide these programming models. However, when it comes to support OppNets, only few systems have been designed to provide the Web services programming model, whilst no other systems appear to be suitable to provide the other programming models for these environments, given the different requirements they impose.

While some of the already-existing middleware systems can deal with a few problems posed by OppNets, these systems do not provide a complete solution. For example, some middleware systems provide ways of dealing with the message loss and/or long transmission delays caused by link disconnections. However, they are predicated on the assumption that link disconnections are exceptional events, which is not the case in OppNets. It is worth taking into consideration that middleware systems that have to cope with some events occurring very rarely are very differently designed from those for

MAJOR CATEGORIES	EXAMPLE MIDDLEWARE SYSTEMS
Distributed objects	RM-ODP CORBA Java RMI
Distributed components	Fractal OpenCOM SUN EJB CORBA Component Model JBoss
Publish-subscribe systems	JMS CORBA Event Service Scribe
Message queues	JMS Websphere MQ
Web services	Apache Axis2 The Globus Toolkit
Tuple spaces	JavaSpaces

Table 2.1: Categories of programming models.

which the events occur very frequently. Therefore, these systems are going to fail once reused in the context of OppNets.

As a result, new middleware systems should be specifically designed for OppNets to provide the aforementioned programming models in these environments while coping with their major challenges.

2.4.3.1 Programming Models Selection Criteria

A fundamental question can be raised about the appropriateness and the suitability of providing the totality of the aforementioned programming models in OppNets. Models that are inherently asynchronous represent good candidates for OppNets, which is not the case of inherently synchronous models.

Eugster et al., have defined three dimensions of decoupling (i.e., asynchronicity) [47]:

1. Space decoupling: interacting endpoints do not need to know each other.
2. Time decoupling: interacting endpoints do not need to be actively participating in the interaction at the same time.
3. Thread decoupling: a caller is not blocked while waiting for some results from the callee.

A programming model can be provided in OppNets on the condition that the following conditions are met:

1. the programming model is inherently asynchronous on some levels (space, time or thread).

2. the redesign, when needed to make it well-adapted for OppNets, will not infringe its original semantics.

At this point it is worth stating that totally synchronous programming models, that imply a strong space, time and thread synchronization, are not appropriate for OppNets. Examples of such synchronous programming models are distributed objects model and distributed components model, which are based on synchronous remote invocations techniques: they evolve and extend Remote Procedure Call (RPC) by adding the concepts of object-orientation/component-orientation, respectively.

The underlying aim of RPC is to transparently extend method invocation to a distributed context so as to hide remote invocations behind a procedural interface, which is based on local procedure calls. The synchronous nature of RPC implies a tight coupling on three different levels: space, time and thread. First, the caller has to know the exact identity and the address of the particular callee (tight space coupling). Second, the caller and the callee must co-exist and be able to synchronize in order to communicate with each other (tight time coupling). Finally, when a procedure is called, the caller relinquishes control to the callee and itself resumes control only upon the return of a result (tight thread coupling).

Some RPC solutions have employed a variety of techniques in their attempt to make the RPC protocol as asynchronous as possible. According to Wade, these mechanisms have fundamentally compromised the behavior of RPC protocol and infringed its original semantics [48]. Moreover, the total distribution transparency required by RPC becomes infeasible using these mechanisms. By considering these issues, it can be determined that programming models based on RPC (or some equivalent systems) are not appropriate for OppNets.

To summarize, the set of programming models that are appropriate and suitable for OppNets are: publish-subscribe, message queues, tuple systems and Web services.

2.4.3.2 Web Services in OppNets

In the context of OppNets, few middleware systems have already been proposed to support the Web services programming model in OppNets, and one of these middleware systems has already been designed in our team (CASA team in IRISA laboratory). Hence, the Web services programming model is not the focus of this dissertation. This section is dedicated to shed the light on this programming model along with the related efforts conducted to provide it in the context of OppNets.

The Web services programming model seeks to encapsulate software functionalities using standardized wrappers. These wrappers are referred to as *services*. A service can be simply viewed as a task or a function that a *provider* can provide to a *consumer*. Consumers may exploit these services using interfaces that are specified at a high level. The salient elements required for *service provision* are as follows:

Service description It describes a service in an unambiguous, human-readable and machine-comprehensive way.

Service advertisement It advertises a service's description on directory services or directly to other hosts in a network.

Service discovery It carries three main functions:

1. formulating a request that describes the needs of a consumer. This request is formulated in the same manner as a service description.
2. providing a matching mechanism that pairs requests with similar service descriptions.
3. providing a mechanism for consumers to communicate with service providers.

Service invocation It facilitates the use of discovered services. Its functions include transmitting commands from consumers to service providers and receiving the corresponding results.

Team CASA (IRISA laboratory) has designed a middleware system for service provision in OppNets [49, 50]. The architecture of this system is structured in two distinct layers. The lower layer is a communication middleware, which ensures the content-driven dissemination of data over OppNets. The upper layer is a high-level service layer that is in charge of all service oriented processing, enabling discovery and invocation interactions between providers and consumers. The service provision leverages the content-based publish/subscribe functions provided by the communication layer. No global service directory is maintained. Instead, a peer-to-peer approach is used in which each provider proactively advertises its services by the way of publication and consumers are informed about the existence of the services they need through subscriptions.

This middleware system has been fully implemented in Java and a number of experiments have been conducted in an OppNet composed of a few devices. The performance of this system is also evaluated using the MADHOC simulator [51] in order to assess its performance in larger configurations. The obtained results demonstrate the feasibility of service provision in OppNets, and confirm the ability of this middleware system to satisfy clients over these environments. Details about the middleware and how it has been tested can be found in [49].

It is worth stating that no assumptions are made regarding the mobility of mobile devices in the above-mentioned middleware system. However, in the literature one can find other works that target a special kind of OppNets that rely on social interactions between human beings carrying mobile nodes, that act as both consumers and providers of services [52, 53]. These works exploit information about the mobility of nodes in an OppNet. Such information allows mobile nodes to estimate when nodes will be encountered, and thus the intercontact time between them. By doing so, the authors of these works propose middleware systems enhanced with some analytical models whose main aim is to minimize the time required to provide composite services. Such middleware systems could not work in the kind of OppNets targeted in this dissertation, in which the mobility model of mobile nodes is not known in advance and the physical carriers are not only human beings carrying laptops or smartphones; they could be for example animals carrying sensors. Along the same lines, Pitkänen et al. provide a service delivery platform called

SCAMPI (Service Platform for Social Aware Mobile and Pervasive Computing) [54]. The SCAMPI architecture leverages human social behavior to enable service provisioning in special kind OppNets that exploit the human social characteristics. Here again, the social awareness required in SCAMPI cannot be satisfied in the kind of OppNets targeted in this dissertation.

2.5 Conclusion

This chapter provided some background about OppNets along with the different problems imposed by such networks. It also introduced the opportunistic computing paradigm in OppNets, supplemented by several real-life scenarios. The chapter finishes by highlighting a set of major types of high-level programming models, that is needed as a basis for opportunistic computing. In the following chapters, this set of programming models, except the Web service programming model, is provided in OppNets through middleware systems, which are specifically designed for these networks so as to fully cope with their major problems.

Part II

Proposal

3 Communication middleware

Contents

BUILDING any application for OppNets requires some communication middleware system, using which mobile hosts can collaborate in a peer-to-peer manner to ensure message transportation, while dealing with high latency and link disruptions.

This chapter introduces the communication middleware systems that are openly-distributed to support communication in OppNets. Then, it illustrates the importance of content-centric networking in the context of OppNets.

3.1 Communication Middleware in OppNets

Although many communication protocols for OppNets have been proposed during the last decade, most of these protocols have only been described in papers as abstract algorithms, and tested using pseudo-code in simulators. Only a handful of these protocols have been actually implemented in middleware systems (and can thus be used in real conditions), but only a couple of these systems are openly-distributed and are thus accessible to developers: DoDWAN, DTN2 and Haggle represent well-known communication middleware systems in this domain. They are introduced in this section.

3.1.1 DoDWAN

DoDWAN *(Document Dissemination in mobile Wireless Ad hoc Networks)*, is a communication middleware system designed in IRISA laboratory (the CASA team) [55], and is now distributed under the terms of the GNU General Public License[1]. This section presents a detailed look at the main architectural design concepts of DoDWAN, along with some examples that describe its functionalities. The main aim is to better understand the basic concepts behind DoDWAN, and the way it supports communication in OppNets.

[1] `http://www-casa.irisa.fr/dodwan/`

```
id= "ff789"
destination_id= "ChatRoom1"
destination_group= "teacher"
date= "Fri Feb 13 20:54:03 CET 2015"
deadline= "Mon Mar 16 20:54:03 CET 2015"
language= "English"
```

(a)

```
destination_id= "ChatRoom.*"
language= "English|Chinese|German"
```

(b)

```
destination_id= "ChatRoom1"
destination_type= "student"
```

(c)

Figure 3.1: Examples of message descriptors and message selector

Content-Based Networking DoDWAN is a communication middleware system, that supports content-based opportunistic message dissemination in OppNets.

In content-based networking, information flows towards interested receivers rather than towards specifically set destinations. This approach notably fits the needs of applications and services dedicated to information sharing or event distribution. It can also be used for destination-driven message forwarding, though, considering that destination-driven forwarding is simply a particular case of content-driven forwarding where the only significant parameter for message processing is the identifier of the destination host (or user).

Note that content-based networking should not be confused with multicast networking, knowing that in both of them messages are generally addressed to several receivers. Content-based networking is actually more flexible than plain multicast. For one thing, content-based communication does not require that specific channels or groups be identified prior to any actual transmission. Besides, the criteria that determines whether a particular message should be received by one or another receiver can be different on each receiver.

Messages Messages represent the basic entities in DoDWAN. They are composed of two parts: a descriptor and a payload. The payload is simply perceived as a byte array. The descriptor is a collection of attributes expressed as *(name, value)* tuples in order to characterize the content of the corresponding message, as illustrated in Figure 3.1a. These attributes can be defined freely by the developers of application services built on top of DoDWAN.

The only exceptions to this rule are a message identifier and a deadline, that must systematically appear in any descriptor. The identifier must be unique, for it allows DoDWAN to differentiate messages while detecting duplicate copies of the same message. The deadline is meant to specify how long a message should be allowed to disseminate in the network, and therefore how long copies of this message should be stored by mobile hosts in their local cache.

Publish/Subscribe Interfaces DoDWAN provides higher-layer services with a publish/subscribe API.

When a message is published by a local application service, it is simply put in the local

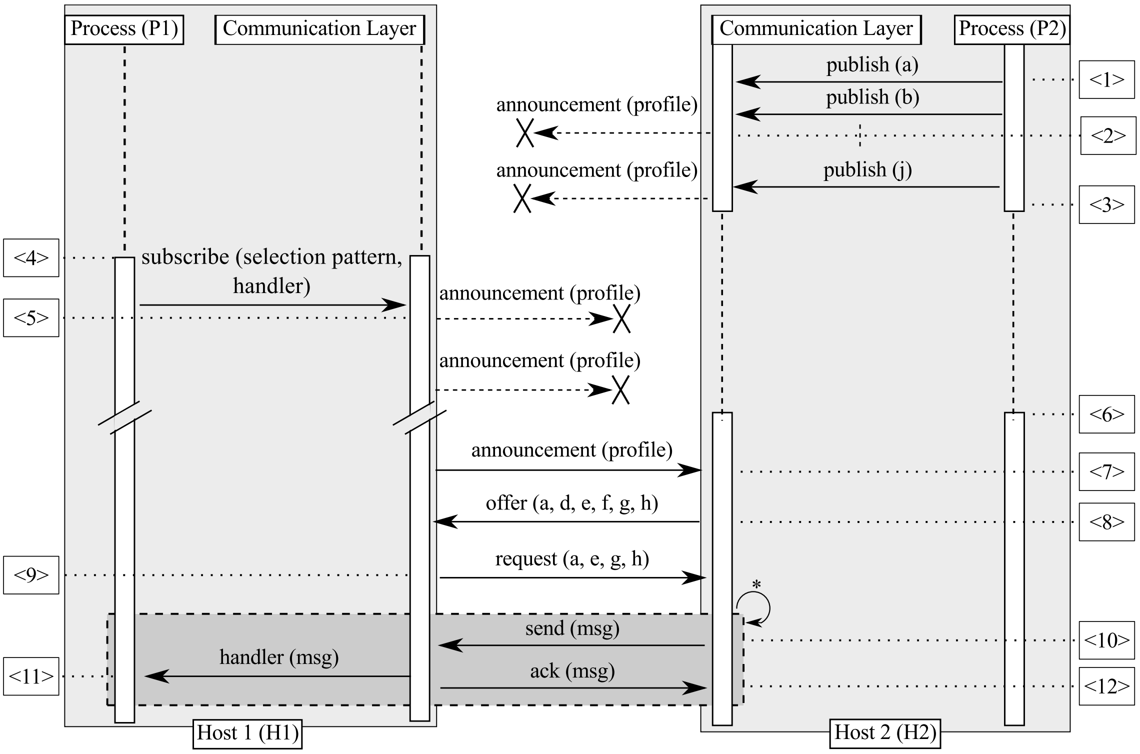


Figure 3.2: Interaction diagram between two hosts exchanging a message using DoDWAN

cache maintained by DoDWAN. Afterwards each radio contact with another host will be an opportunity for DoDWAN to transfer a copy of the message to that host whenever it is interested.

In order to receive messages an application service must subscribe and provide a *selection pattern* as *(name, regular expression)*, that characterizes the kind of messages it would like to receive. It is worth noting that a selection pattern is expressed just like a message descriptor, except that the *value* field of each attribute contains a regular expression. Figure 3.1b shows a selection pattern, which would for example match the message descriptor shown in Figure 3.1a. In contrast, the selection pattern shown in Figure 3.1c would not.

The selection patterns specified by all local application services running on the same host define this host's *interest profile*. DoDWAN uses this profile to determine which messages should be exchanged whenever a radio contact is established between two hosts.

Opportunistic Networking Message routing in OppNets has already justified a fair amount of research. Most of the algorithms proposed rely on more or less constrained variants of the epidemic dissemination scheme, as defined in [56] and [57]: whenever a message is sent, several copies of this message are actually produced so these copies can propagate separately in the network, each copy being carried by a distinct mobile node. This approach is clearly a costly one but it helps preventing message loss, which can occur because of transmission failures (when a message is transmitted wirelessly between two neighbor nodes), or because the carrier of a copy may unexpectedly disappear from the network. Many solutions have been proposed to keep the cost of epidemic routing at a reasonable level, using heuristics that basically aim at reducing the number of carriers for each message, and also sometimes at selecting the "best" carriers for each message. Some solutions rely on probabilistic or semi-probabilistic heuristics [58, 59, 60], or take into account the context of each mobile node [61, 62, 63]. Others assume that mobile nodes are carried by human beings, whose social interactions can be captured and used to drive message forwarding by predicting how people move or meet [64, 65], or by identifying what communities each person belongs to [66, 67, 68].

DoDWAN makes no assumption about the carriers of mobile nodes. These carriers could thus be human beings, but they could as well be vehicles, robots, animals, or any combination of these. The interaction scheme implemented in DoDWAN takes inspiration from the Autonomous Gossiping (A/G) algorithm [57], which itself defines a selective version of the epidemic routing model proposed in [56]. Each host periodically broadcasts an announcement in order to inform its neighbors (if any) about its identity and interest profile. By sending such an announcement periodically, a node informs its neighbors about its presence and about the kinds of messages it is interested in. Conversely, by receiving similar announcements a host discovers its neighbors, and learns about their own interest profiles. By matching its neighbors' profiles against the descriptors of the messages it maintains in its cache, a host can select descriptors of messages, that might be of interest to at least one of its current neighbors. It can thus build a catalog containing these descriptors, and incorporate this catalog in its next announcement. Upon receiving such a catalog, each host matches the descriptors it contains against its own interest profile in order to identify messages, that match this profile and that are not already present in its local cache. If such messages are identified, then a request for these messages is sent to the announcer, which complies by sending the missing messages through a series of *send+ack* cycles on the radio channel. Finally, every time a host receives a message it has requested, this message is put in the local cache so it can later be proposed to other hosts met while moving in the network.

Figure 3.2 illustrates the gossiping between two hosts *H1* and *H2* as they exchange messages. In this example we can assume that *H1* and *H2* are within mutual transmission range, but that they are sometimes in suspend mode (though not necessarily at the same time). The same behavior could be obtained with these hosts moving and getting in a transient radio contact, though.

An application process *P2* on *H2* first publishes several messages *a, b, c... j* (label <1> in Figure 3.2). *H2* then starts announcing its presence periodically, with an interest profile that includes the published messages (label <2>).

Since *H2* has no neighbor at that time, these messages are simply put in the host's local cache. *H2* then enters suspend mode for a while (label <3>). *H1* is started (for the first time, or resuming from suspend mode), and an application process on that host subscribes to receive messages that match a specific selection pattern (label <4>). *H1* then begins announcing its presence, with an interest profile that includes the pattern specified by process *P1* (label <5>). These announcements are not received by any host, though, since *H1* currently has no neighbor.

H2 resumes (label <6>), and shortly after that it receives an announcement broadcast by *H1* (label <7>). *H2* thus discovers this neighbor, and it simultaneously learns what kind of messages *H1* is interested in. Based on *H1*'s interest profile, *H2* can check its local cache and identify messages whose descriptors match this profile. *H2* can thus sends an offer to *H1* (label <8>). Upon receiving this offer *H1* determines which of the offered messages it would actually like to receive (*i.e.,* messages that are not already in its own local cache), and sends a request accordingly to *H2* (label <9>). For each message thus requested, *H2* sends the message to *H1* (label <10>) and waits for an acknowledgement before sending the next message (label <12>). Each message received by *H1* is put in the local cache, and the handler of the application process is called in order to process the message (label <11>). Although this is not represented in Figure 3.2 for the sake of clarity, this interaction pattern involving *H2* providing messages to *H1* is also run symmetrically: *H1* may likewise provide *H2* with messages that match its own interest profile. Moreover, *H1* and *H2* may simultaneously be interacting with several other neighbor hosts. Finally, the interaction procedure between two hosts is stateless and can thus be interrupted at any time: both hosts will simply detect that they have lost a neighbor, and act accordingly.

More details about this interaction schema and about how it performs in real conditions can be found in [55].

Handling Connectivity Disruptions To handle the high level of connectivity disruptions expected between hosts in OppNets, interactions between neighbor hosts rely on an opportunistic scheme rather than on a strict transactional scheme. No session –and especially no TCP session– is ever established between neighbor hosts. Indeed, mobile hosts running DoDWAN only interact by exchanging control and data messages encapsulated in UDP datagrams, which can themselves be transported either in IPv4 or IPv6 packets.

Each host only maintains soft-state information about its neighbors. Thus, whenever a host broadcasts an announcement, for example, some of its neighbors may fail to receive this announcement, without ever compromising either the sender or any potential receiver. Likewise, whenever a host requests a message and fails to obtain this message, it simply waits until it can get another chance to grab this message (either from the same neighbor, or from a different one).

Large messages are additionally segmented so that each fragment can fit in a single UDP datagram. Fragments of a large message all contain a copy of the original message's descriptor, so they can propagate independently in the network and be reassembled only on destination hosts.

Data Mules and Altruistic Carriers As a general rule, a mobile host that defines a specific interest profile is expected to serve as a *data mule* for all messages that match this profile. To this end, it carries these messages for a while in its local cache so they can be transferred later to other interested receivers. Using the system model introduced in Section 2.2.2, messages that match an interest profile p can be carried by a subset of nodes $\mathcal{V}_P \subseteq \mathcal{V}$, which consists of all nodes in $\mathcal{V}$ whose interest profile is p (or larger than p). Possible journeys for copies of a message that matches p are defined in $\mathcal{G}_P = (\mathcal{V}_P, \mathcal{E}, \mathcal{T}, \rho, \psi)$.

Yet, a host can also be configured so as to serve as an *altruistic carrier* for messages that present no interest to the application services it runs locally. This behavior is optional, though, and it must be enabled explicitly by setting the configuration parameters of DoDWAN accordingly. Using our system model, any host $u \in \mathcal{V} \setminus \mathcal{V}_P$ can serve as mobile carriers for messages matching p, and thus belong to a subset $\mathcal{V}_{\mathcal{C}(P)}$(carriers for the interest profile p) such that $\mathcal{V}_P \subseteq \mathcal{V}_{C(P)} \subseteq \mathcal{V}$. Any message matching p, whatever its sender $u \in \mathcal{V}_P$, is assumed to disseminate thanks to nodes in $\mathcal{V}_{\mathcal{C}(P)}$, and eventually reach all or some of the nodes in $\mathcal{V}_P$. This implies that when a message m is sent by $u \in \mathcal{V}_P$, there actually exist journeys in the TVG $\mathcal{G}_{C(P)} = (\mathcal{V}_{C(P)}, \mathcal{E}, \mathcal{T}, \rho, \psi)$ that lead from u to nodes in $\mathcal{V}_m$, with $\mathcal{V}_m \subseteq \mathcal{V}_P$ and $\forall v \in \mathcal{V}_m$, $\exists u \rightsquigarrow v$. The reason why only *some of the nodes* in $\mathcal{V}_P$ are considered here is a consequence of the epidemic routing model, which cannot guarantee that each message eventually reaches all its possible recipients.

Resource Management As mobile hosts exchange messages, caches grow and storage may become insufficient. Along the same lines, carrying many (and sometimes outdated) messages slows down DoDWAN and may cause unnecessary dissemination. DoDWAN avoids cache overloading and network congestion by setting a *deadline* for each message. Whenever a message gets out of date, all copies of this message are removed from caches, so it stops its dissemination in the network. There is no need for an exact clock synchronization of the nodes to ensure a coherent management of the messages deadlines: some difference between two nodes clocks is acceptable and does not jeopardize the whole mechanism.

The dissemination of a message can also be *canceled* explicitly on a mobile node. Once a message is canceled, the node does not propose it to any neighbor anymore, and if conversely a neighbor actually offers to provide this message, this neighbor is notified that it too should cancel the message whose dissemination is not required anymore. This approach is sometimes referred to as *network healing* in the literature. It is an effective way to limit the cost of epidemic routing in an OppNet, using either *Passive Cure* [69, 70] or *Active Cure* techniques [71]. Both mechanisms, i.e., deadline and network healing, if used wisely can help reduce the cost of epidemic routing in an OppNet.

3.1.2 DTN2

DTN2 is the current reference implementation for the DTNing architecture addressed by the Delay-Tolerant Networking Research Group (DTNRG). The implementation designed by this group is an evolution of another implementation originally designed for the Interplanetary Internet, which focused primarily on the issue of deep space communication in long-delay environments. The DTN2 implementation provides a set of mechanisms suitable for delay-tolerant host-centric networking, with the attendant need for temporal decoupling and spatial coupling.

Hosts using this architecture are referred to as “nodes” and identified by a DTN-specific naming syntax, Endpoint Identifiers (EID) [10]. Nodes communicate by exchanging data blocks named "bundles".

Conceptually, the DTNing architecture operates as an overlay network on top of convergence layer adapters. Several convergence layers are defined, so bundles can for example be transported between two nodes using TCP sessions, UDP datagrams, plain files, or any other convenient transport protocol.

DTN2 is meant to ensure end-to-end delivery of bundles across the entire network in a store-and-forward manner. When two nodes are within range and have an established connection between them (a.k.a, a link), they can start exchanging bundles. In DTN2, routers control which bundles are sent over which links.

DTN2 includes a number of internal routers, such as: static, flood, epidemic, PRoPHET routing, etc. DTN2 also includes support for external routers. External routers run as separate processes and communicate with DTN2 by sending and receiving XML messages over a multicast socket. DTN2 sends XML event messages to a multicast socket, from which the external router reads. The external router then processes the message, and may or may not decide to send back an XML request message, prompting DTN2 to take action.

DTN2 supports reliability between adjacent nodes, by custody transfers. When a bundle is transferred, using the custody transfer, to an adjacent node and the destination node accepts it, this node is meant to send back a bundle acknowledgment. If this acknowledgment is not received before a certain time expiration, the sender will retransmit the bundle. The bundle will remain in the custodian node until another node accepts its custody or the bundle lifetime expires. This mechanism does not guarantee end-to-end reliability and can only be done if the source of communication requests the custody transfer.

DTN2 represents the only openly distributed host-centric communication middleware in the context of challenged networks. Hence, it can easily be deployed in order to add host-centric services to current systems. However, this system has not been primarily designed to target OppNets presenting short, unpredictable radio contacts between mobile hosts. Hence, it is still unclear if it could run satisfactorily in such conditions.

3.1.3 Haggle

Haggle[2] is a content-sharing system for mobile devices, allowing users to opportunistically share content items and interests without the support of infrastructure [72]. Haggle implements a ranked search to decide what content to exchange and in what order. The search matches the content stored locally on a node against the interests of the other users that the node has collected. A mobile device can additionally behave as a benevolent carrier for messages it is not interested in, though an optional content delegation mechanism.

Each user has a local data store in order to store and index the content items it shares based on their metadata. Metadata represent the address of a content item; similar attributes between two items are represented in a relation graph. The weight of a relation depends on the number of mutually shared attributes. The content of this relation graph is updated with content and the interests of other users as they are encountered.

The information dissemination in Haggle relies on user sharing interests. Haggle disseminates interests in the network to facilitate push-based rather than pull-based dissemination. To this aim, each node shares a *node description* that carries, among other information, the interests of the node (in the form of the attributes) and a list of the content items it stores in its local data store. Received node descriptions are stored in the

[2] `http://haggleproject.org/`

data store of the receiving node. These node descriptions are then used to query the local data store for matching content items, which are then pushed back in the network. Node descriptions can further propagate to third-party nodes that repeat this search, pushing back content items in the network.

Haggle is mostly written in C++ and API wrappers have been provided for C# and Java. Haggle is openly available to developers with a few simple applications like photo sharing to demonstrate how to use it.

The way node descriptions are dealt with in Haggle may yield significant overheads in OppNets. First, these node descriptions must be disseminated network-wide in order to enable the push-based dissemination in Haggle. Second, a node description must encode all the content items stored in the originating node in order to handle content duplicates (i.e., to prevent the same content item from being pushed several times to the same target nodes). Third, when a node has successfully sent a content item to another node, it adds the content item to its copy of the receiver's node description to avoid sending it again. The aforementioned issues significantly increases the size of node descriptions which, in turn, causes extra overhead in terms of additional network and storage load.

According to Nordström et al., the current implementation of Haggle leverages Prophet [73] and epidemic-style protocol for content delegation algorithms [72]. However, Daly et al. have demonstrated that Prophet may fail in sparse networks due to the low connectivity of the sending and the receiving nodes, which is exactly the case in OppNets [74]. Daly et al. have also demonstrated that epidemic algorithms cause non-negligible overhead in OppNets, thus increasing the overhead caused by Haggle once used in these environments. Hence, the current implementation of Haggle is not adequate for use with OppNets, and it must be extended with the support of other algorithms, e.g., the Autonomous Gossiping (A/G) algorithm supported by DoDWAN, in order to be well-adapted for these environments.

3.2 The Importance of Content-centric Networking in OppNets

In the literature, the proposed data exchange policies may be classified according to two main paradigms: host-centric and content-centric networking. In content-centric networking, information flows towards interested hosts rather than towards specifically set destinations. That is, the flow of information is interest-driven rather than destination-driven [75].

Content-centric networking can provide flexible systems in the context of dynamic environments. Receivers specify the kind of information they are interested in, without regard to any specific source. Senders simply send information in the network without addressing it to any specific destination. The relevant importance of using content-centric networking in the context of ad hoc networks has been presented in [76]. It can be summarized as follows: content-centric networking relieves the burden on application developers to deal with a variety of communication modalities. In fact, developers need only to know "what" information they wish to consume, but do not really care about "where" the information resides in the network, nor via which protocol, information arrives. Thus, there is no reason for applications developers to be dealing with host names, addresses and byte-streams; instead, they should be dealing directly with information and application data units. Content-centric networking helps developers to spent very little design time on how to handle the mutual interactions between their applications. Adding a communication participant on any content-centric based system becomes an easy, almost trivial task. Such features make content-centric networking a better solution for systems in a wide variety of dynamic environments.

Middleware systems presented in this dissertation rely on content-centric networking (provided by DoDWAN), which is suitable for OppNets with the attendant need for spatial and temporal decoupling.

3.3 Conclusion

In OppNets, a very limited number of communication protocols has been actually implemented in middleware systems (and can thus be used in real conditions). Yet, a couple of these middleware systems are openly distributed and are thus accessible to developers: DoDWAN, Haggle and DTN2. This chapter briefly reviewed these communication middleware systems. The middleware systems presented in this dissertation currently leverage DoDWAN. Yet they could theoretically be implemented above any other communication system, provided this system could operate satisfactorily in OppNets.

4

A Java Message Service Provider for Opportunistic Networks

Contents

4.1 Introduction

DEPLOYING applications for OppNets becomes possible thanks to the work carried out on communication protocols. However, all application models are not suitable for this type of networks. The characteristics of OppNets should be taken into account from the application design stage, dealing with asynchronicity, long transmission delays, message loss, erratic availability of distant hosts. Extensive research has been conducted in order to ease the development and deployment of distributed applications for traditional fully-connected networks. The conducted research usually relies –explicitly or implicitly– on the assumption that a reliable end-to-end transport layer is always available, providing reliable communications between application components. In OppNets, contacts between devices are intermittent and are hardly predictable. Additionally, because of the sparse and irregular distribution of mobile devices, neither end-to-end connectivity nor transmission delays can be guaranteed. Efforts must be pursued in the context of OppNets.

There is a wide acknowledgment that the middleware approach can facilitate the construction of distributed applications [77, 78]. A middleware system can be layered between networks and application components, so as to provide high-level programming models that make it easier for programmers to deal with distributed components. As a result, programmers can focus on the specific purpose of their application.

According to the categorization proposed in Chapter 2, one can observe that several high-level programming models can be supported by the same middleware solution. Indeed, the publish-subscribe programming model is a sibling of the message queue programming model. Both models are generally supported by a larger Message-oriented Middleware (MoM) system, e.g. Java Message Service (JMS). This chapter sheds the light on the JMS as a general-purpose MoM system, in which both publish-subscribe programming model and message queues programming model are supported, offering developers the choice of one-to-many or one-to-one variants of communication models, respectively.

Our objective in this chapter is to provide the publish-subscribe programming model and the message queues programming model in the context of OppNets through a disruption-tolerant JMS provider, so as to enable pre-existing or new JMS-based applications to be easily deployed in OppNets.

This chapter introduces JMS, describes its key features and explores how to make it disruption-tolerant so as to be able to run in OppNets. The design and the implementation of a disruption-tolerant provider for JMS are then discussed, and experimental results obtained during real-life measurement campaigns are presented.

4.2 Background

The Java Message Service (JMS) is a Message-oriented Middleware (MoM) standard, that allows application components based on the Java 2 Platform Enterprise Edition (J2EE) to communicate in a loosely-coupled, reliable, and asynchronous manner.

The JMS specification, in general, covers more than just message passing. It includes services for translating data, broadcasting notifications to multiple programs, locating resources in the network, prioritizing messages and requests, supporting atomic operations (transactions), supporting message redelivery, supporting acknowledgments, and extensive debugging facilities.

JMS, as a MOM product, does not assume the system has a reliable transport layer underneath, and tries to address the problems that may occur when dealing with an unreliable transport layer.

The JMS specification defines two communication models[1]: point-to-point, and publish-subscribe.

The point-to-point model is built around the concept of JMS queues. A *queue sender* sends a message to a specific queue, in which it wishes the message to be placed. A *queue receiver* can then receive this message asynchronously. This model provides a *one-to-one* communication model. In other words, a given queue may have multiple senders and multiple receivers, but each message can be consumed by only one receiver. This means that some mechanism is required to decide which receiver candidate will be the actual

[1]Communication models are referred to as high-level programming models according to the terminology used in this dissertation.

receiver of a given message.

The publish-subscribe model is based on the concept of topics, that can be subscribed to by *topic subscribers*. Messages are published to a topic by *topic publishers* and they are then received asynchronously by the corresponding *topic subscribers*. Each message may thus be consumed by multiple subscribers. This model complements the point-to-point model, in that it provides a *one-to-many* communication model.

It is worth noting that the mechanism by which the message is transmitted is completely hidden from the application programs.

4.3 Motivation for Providing JMS for OppNets

JMS can be provided for OppNets because of its inherently asynchronous nature on the three dimensions: space, time and thread, as discussed in Section 2.4.3.1. Furthermore, its popular specification eases application development in these environments. This section sheds the light on the important features of JMS that enable it to operate asynchronously in any network.

Space Asynchronicity In JMS, messages are delivered through an intermediary (i.e., queue or topic), decoupling the sender from the receiver. As a result, the interacting sender/receiver do not need to know each other in order to communicate. This feature is known as space asynchronicity [47], allowing the sender to send a message to an intermediate topic/queue, while the system delivers the message to corresponding receivers.

Time Asynchronicity Using an intermediary in JMS, the sender/receiver do not need to be actively participating in the interaction at the same time. The JMS provider delivers the message when conditions are met. In the literature, such a feature is usually referred to as time asynchronicity [47], which indeed is a highly desirable feature for a system in the context of OppNets.

Thread Asynchronicity Using JMS, the sender can continue to carry out other task as soon as it completes sending a message, without being blocked waiting for the message to be received by the receiver. This feature, which is known as thread asynchronicity [47], represents another important feature in OppNets since the sender cannot be blocked forever waiting for a response; knowing that the waiting period cannot be accurately determined in the context of these challenged environments.

Standard Specification In order to provide efficient messaging systems, implementing the well-known JMS interfaces, which are largely used by developers, is actually preferred over defining new specific interfaces, which would result in an exotic system known to very few developers.

To summarize, the asynchronous nature of the JMS system along with its standard specification represent the main motivations for providing a JMS implementation for OppNets.

4.4 Programming Model

JMS is a vendor-neutral Java messaging standard, which defines a common set of interfaces including the associated semantics. JMS allows distributed Java programs to communicate indirectly [41].

The JMS specification defines neither how JMS messages are transported, nor how JMS destinations are implemented within JMS providers. Indeed, it only defines the ways by which messages should be created, produced and consumed. Because of the lack of a standard implementation, each vendor proposes its own JMS provider, along with the associated management tools. Each JMS provider supplies the user with an appropriate transport system for a particular deployment environment. There are several JMS providers available like OpenJMS [2], HermesJMS [3] and JORAM [4].

This section briefly sheds the light on the JMS terminology and concepts, which will be used along this chapter.

4.4.1 Basic JMS Terminology

Using the JMS terminology, one can distinguish between the following key roles:

JMS provider: a system that implements the JMS specification. A wide variety of JMS providers are now available, and the system developed in this chapter is one of them, each with particular perspectives.

JMS client: a Java application that uses JMS. A JMS client can either produce or consume messages, thus acting as a *JMS producer* or as a *JMS consumer*, respectively. A single JMS client can be both a producer and a consumer.

A JMS producer produces messages, which can then be consumed *asynchronously* by JMS consumers. The producer and the consumers do not have to be coupled in order to communicate.

JMS application: a business system composed of several JMS clients/providers.

JMS message: an information item, that is used for communication between JMS clients. A JMS message has three parts: a header, properties, and a body.

The JMS message header contains fields used by both clients and providers for the identification, control and delivering of messages. JMS supports two delivery semantics, through the so-called persistent and non-persistent delivery modes. A non-persistent message should be delivered in a best-effort mode. Conversely, a persistent message must be delivered in a guaranteed mode. The desired delivery mode for a message is specified by setting the field *JMSDeliveryMode* accordingly in the message's header.

[2] `http://openjms.sourceforge.net/`
[3] `http://www.hermesjms.com`
[4] `http://joram.ow2.org/`

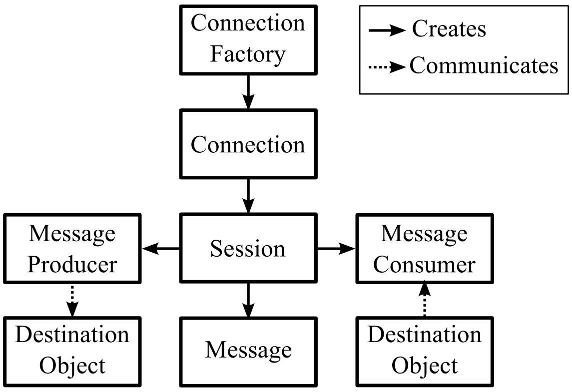


Figure 4.1: Overview of JMS object relationships

Properties are extra header fields, that qualify the message content. They can be used by clients to filter messages via message selectors. Selection criteria cannot reference the message body, though, as this body is opaque and untouched by the system.

The message body carries the message content, which can take several forms: text, map, bytes, stream and serializable objects.

JMS destination: a network-independent intermediate object supporting indirect communication in JMS. It represents the target for which messages are produced or the source from which messages are consumed. A JMS destination can either be a *JMS topic* or a *JMS queue*.

4.4.2 Programming with JMS

An overview of the different JMS objects and their relationships is shown in Figure 4.1.

The connection factory represents the start point for a JMS client to interact with the given JMS provider. It allows the JMS client to create an actual connection with the provider. The resultant connection is a logical channel, which is then mapped onto a physical channel by the underlying implementation. Connection can then be used to create one or more sessions, which play a key role in the operation of JMS: each session runs in a single-threaded context, which groups a series of operations involving the creation, production and consumption of JMS messages related to a logical task. A multi-threaded JMS application must create different JMS sessions.

Code. 4.1 presents a simple JMS application whose sole purpose is sending and receiving text messages between two users (i.e., private chatting). The import statements are not shown. This code has been tested using the OpenJMS implementation [79]. The message queues model is chosen as this is intrinsically a one-to-one application, with a client producing a message targeted towards another client.

The starting point for this JMS application implies the acquisition of a connection factory (line 14) and a queue location (line 16), the creation of a connection (line 18) and then the creation of a session (line 20). The session is then used to create a message (line 25) and a message producer (line 27). The client can now use this message producer

Code 4.1 A JMS application: Private Chat

```java
public class HelloWorldMessage{
    ConnectionFactory myConnFactory;
    Connection myConn;
    Session mySess;
    Queue myQueue;
    AdminTools admin_tools;
    Context jndi;

    void initialization() throws Exception{
        admin_tools = AdminToolsSingleton.getInstance();
        // Obtain a JNDI connection
        jndi = ContextSingleton.getInstance();
        // Lookup a ConnectionFactory administered object.
        myConnFactory= (ConnectionFactory) jndi.lookup("HelloWorldConnectionFactory");
        // Lookup a Queue Destination administered object.
        myQueue = (Queue) jndi.lookup("myqueue@benchi-Qosmio-G50");
        // Create a connection to the Message Queue Service
        myConn = myConnFactory.createConnection();
        // Create a session within the connection.
        mySess = myConn.createSession(false, Session.AUTO_ACKNOWLEDGE);
    }

    void send() throws Exception{
        // Create a text message.
        TextMessage myTextMsg = mySess.createTextMessage();
        // Create a message producer.
        MessageProducer myMsgProducer = mySess.createProducer(myQueue);
        myTextMsg.setText("Hello World");
        System.out.println("Sending Message: " + myTextMsg.getText());
        // Step 9: send the message to the queue
        myMsgProducer.send(myTextMsg);
    }

    void receive() throws Exception{
        // Create a message consumer.
        MessageConsumer myMsgConsumer = mySess.createConsumer(myQueue);
        // Start the Connection created in step 3.
        myConn.start();
        // Receive a message from the queue.
        Message msg = myMsgConsumer.receive();
        // Retreive the contents of the message.
        if (msg instanceof TextMessage) {
            TextMessage txtMsg = (TextMessage) msg;
            System.out.println("Reading: " + txtMsg.getText());
        }
    }

    void close () throws Exception{
        // Close the session and connection resources.
        mySess.close();
        myConn.close();
    }

    public static void main(String[] args) throws Exception{
        HelloWorldMessage jms= new HelloWorldMessage();
        jms.initialization();
        // Sender-side code
        if (args[0].equalsIgnoreCase("sender"))
            jms.send();
        // Receiver-side code
        else if (args[0].equalsIgnoreCase("receiver"))
            jms.receive();
        jms.close();
    }
}
```

to send the already-created message to the queue (line 31). The corresponding code for the receiver-end JMS client has a similar structure. However, the session is used to create a message consumer (line 36), which is then used to receive the message from the queue (line 40).

According to the JMS specification, JMS clients cannot create connection factory objects or destination objects. These special objects, called administered objects, can only be created and configured by system administrators. JMS clients can learn about them, as shown in line 14 and 16, using the naming services provided by JNDI (Java Naming and Directory Interface). This is discussed further in the next section.

4.4.3 Understanding Java Naming and Directory Interface (JNDI)

According to the JMS specification, JMS applications learn about the available administered objects through the Java Naming and Directory Interface (JNDI).

JNDI is a Java API for accessing various naming and directory services and different databases [80]. As a vendor-neutral specification, JNDI does not provide its own implementation. It rather indicates the way with which administered objects should be dealt. JNDI is also independent from the underlying directory services, which could be implemented using a server, a plain file, or a database; the choice is up to the implementing vendor.

According to the JNDI specification, any directory service must provide a hierarchical structure, referred to as a *namespace*. JNDI uses this namespace to map names to the corresponding objects. Applications can thus discover names and look any data item or object up via its name. In traditional server-based JNDI implementations, a client creates an administered object via some administration tool and binds it to a central JNDI server. This object then resides there until it is unbound. From then on, applications typically use JNDI to lookup the administered objects they require.

Using JNDI, JMS clients can browse a naming service and obtain references to administered objects without knowing the details of the naming service or how it is implemented. For the JMS application shown in Code. 4.1, obtaining a connection to the JNDI naming service represents the starting point for using JNDI (line 15). It allows the JMS client to access the different administered objects (line 17 and 19).

4.5 Middleware Architecture and Implementation

JMS was primarily designed for systems where clients connect to central servers via traditional networks and this has remained its typical usage scenario. In most implementations of JMS, message producers send messages to topics/queues that are managed by a server. These topics/queues *store* the messages and *forward* them later to the consumers. Furthermore, these implementations leverage a server that provides a directory service so as to allow JMS clients to discover administered objects.

As explained in Section 2 a server-based model is hardly compatible with the characteristics of OppNets. No host can act as a reliable server for all other hosts. A server-less

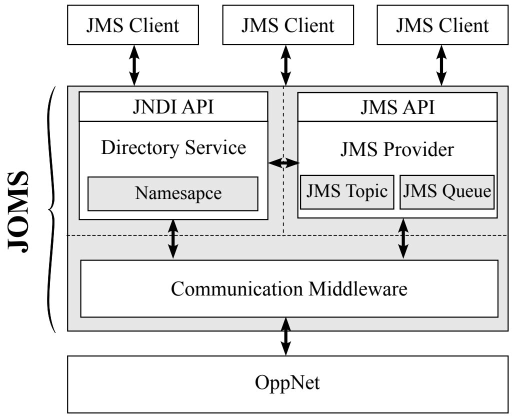


Figure 4.2: JOMS architecture

JMS implementation must thus be developed in order to provide JMS services in OppNets.

JOMS, or Java Opportunistic Message Service, is a JMS provider that was designed along that line. Its architecture is shown in Figure 4.2. JOMS is composed of three basic modules: a communication middleware system, a directory service, and the JMS provider per se. Currently, JOMS leverages DoDWAN for communication services, as described in Chapter. 3. This section gives more details about the two other modules, i.e., the directory service and the JMS provider.

4.5.1 Directory Service (JNDI)

Application components must look up the JMS administered objects they intend to use in a directory service. The JMS specification states that objects discovery must be made through the JNDI API [80]. JOMS implements a subset of the API of the standard JNDI, while preserving the semantics. The interface of the directory service supported by JOMS is shown in Code. 4.2.

Code 4.2 The interface of the directory service supported by JOMS

```
public interface Context{
    public Object lookup(String name);
    public void bind(String name, Object obj);
    public void rebind(String name, Object obj);
    public void unbind(String name);
    public void rename(String oldName, String newName);
}
```

In JOMS, each host maintains a local directory, that acts as a local namespace, from which JMS applications can lookup (i.e., retrieve) administered objects. When an administered object is created, an entry for this object is added to the local namespace.

By delving into the finer details, each entry is characterized by three fields: a type, a name and a deadline, as it is explained further below:

```
entry_type= "destination_object"
destination_id= "ChatRoom1"
destination_type= "topic"
deadline= "Tue Sept 13 19:27:32 CET 2016"
```

Figure 4.3: An example of a administered object

Type: it can be either a JMS connection factory or a JMS destination object. A destination object is further subdivided into two properties: a JMS topic or JMS queue. These properties has been added since the two modes of communication are managed quite differently by JOMS .

Name: it is used by JMS applications to obtain a reference to the corresponding administered object (technically speaking, this is done using the method *lookup*)

An entry in a namespace must be characterized by a unique name. Ensuring name unicity in a server-based configuration over traditional (connected) networking environments does not raise any major issue. Ensuring name unicity in a distributed configuration can be interpreted as solving a distributed consensus problem in a such condition, where system administrators can choose collaboratively the name of each administered object so as to guarantee its name unicity network-wide. Unfortunately, while implementing JOMS, no openly-available consensus implementation was yet available in OppNets and we were still developing our ideas on how to solve consensus in OppNets. As a result, this issue must be dealt with differently using a simpler technique without leveraging consensus.

To this aim, connection factory or topic created on different devices with the same name are considered as being the same objects. Indeed, connections created using that connection factory are all the same and all messages published in that topic will be received by all subscribers. It is up to the administrator to manage connection factory/topic creations while avoiding name ambiguities, which is beyond the scope of this dissertation. In contrast queues must have unique names, as each queue is to be maintained on a single device (as explained in the next section). When a queue is created, the name specified by the user is appended with the local device's identifier, which is provided by the communication layer (this identifier can for example be the MAC address of a local network adapter, the IMEI on a smartphone, or an auto-configured link-local IPv6 address).

Deadline: it represents the expiration date of the entry in order to keep local spaces up-to-date.

An example of an administered object's entry is shown in Figure 4.3. This entry describes a topic with name "ChatRoom1" and deadline "Tue Sept 13 19:27:32 CET 2016".

Once an administered object has been added in a host's local namespace, JNDI advertises this object using the underlying communication middleware. The hosts that receive an advertisement message that pertains to new administered objects (that is, administered objects that are not already available in their own namespaces) update their own

(a) Step 1 (b) Step 2 (c) Step 3

Figure 4.4: Example of a JNDI namespace dissemination

namespaces accordingly.

Let us consider a simple scenario for the sake of illustration. Consider the two mobile hosts carried by human beings shown in Figure 4.4a. The local namespace of the first host contains three administered objects: *queue 1*, *queue 2* and *topic 1*. The second one also has three administered objects: *topic 1*, *topic 4* and *queue 2*. When these two hosts meet, as shown in Figure 4.4b, they exchange information about the content of their respective namespaces. Note that the users carrying these two hosts do not have to be aware of this exchange. Thanks to the communication middleware the gossiping between neighbor hosts is performed opportunistically (and transparently for the users) whenever two hosts meet. Based on this gossiping, each host updates its own local namespace with the new administered objects it has just discovered. Consequently, as shown in Figure 4.4c both hosts eventually present an identical namespace, which contains all four administered objects.

The JNDI API defines primitives to unbind objects from a repository. Disseminating an unbind advertisement for an object in an OppNet does not ensure that the dissemination of the corresponding binding advertisement will effectively be stopped. In practice, unbinding an administered object network-wide represents another issue, which requires solving a distributed consensus in this network, where mobile devices can agree to unbind the administered object network-wide so as to stop disseminating its binding advertisement, accordingly.

Here again, this issue can be dealt with alternatively by leveraging the deadline property introduced in each JNDI entry in order to delete obsolete administered objects. In addition, a refresh mechanism is used to reset the deadline of entries automatically, as long as they are used by application services: whenever a device uses a connection factory or sends/receives a message to/from a specified destination object, the deadline for the corresponding entry is systematically prolonged in this device's namespace. Thus, if an entry is not refreshed for a long time, this entry is finally considered as obsolete and is automatically deleted from the local namespace. This mechanism prevents a host from advertising an out-of-date administered object and ensures that each host maintains its namespace up to date.

4.5.2 JMS Provider

A JMS provider has to support the *publish-subscribe* and the *point-to-point* styles of messaging. This section describes the message model of JOMS and the way it supports the two models of communication.

4.5.2.1 Message Model and Management

As mentioned in Section 4.4.1, a JMS message has three parts: a header, properties, and a body. The header and properties contain fields for the identification, control and routing of messages. JOMS uses those fields to manage the opportunistic dissemination of messages. The message body carries the message content. Its content is ignored by JOMS while matching messages. It is simply considered as a payload.

JOMS introduces a few discrepancies with respect to the standard JMS properties, in order to deal with the nature of OppNets.

By delving into the finer details, while the *JMSExpiration* field is optional according to the JMS specification, it is now mandatory as it is employed to avoid the overloading of radio channels and hosts caches with out-of-date messages. Its value is assigned to a default if not provided by developers.

Furthermore, the *JMSMessageID* property which is optional according to the JMS specification, becomes also required in JOMS, since it is used to organize messages locally. Its value is assigned to a random value if not provided by developers.

Message organization is done using the principal of hashing supported in Java. When a user creates a new message, JOMS applies a hash function on the *JMSMessageID*. The obtained hash code is then used to specify a location on the local cache for storing the message. Similarly, the message is then located and retrieved using its *JMSMessageID*'s hash code.

According to the JMS specification, the *JMSDeliveryMode* and *JMSPriority* express the expected degree of reliability and priority for transmitting messages.

JMS leverages the *JMSDeliveryMode* property in order to allow programmers to specify the expected degree of reliability (guaranteed vs best-effort delivery), through the so-called *persistent* and *non-persistent* delivery modes. On the one hand, persistent messages are stored in persistent storage to be delivered at a later date if the destinations are unavailable. A JMS provider must deliver this kind of message *once-and-only-once*, i.e., it cannot be lost in transit due to a JMS provider failure and cannot be delivered more than once. On the other hand, non-persistent messages do not need to be stored in persistent storage in case of failure. A JMS provider must deliver this kind of message *at-most-once*, i.e., the message can be lost, but can only be delivered once.

Along the same lines, JMS leverages the *JMSPriority* property to express the expected priority level when transmitting messages. Indeed, it defines ten levels of priority values, with 0 as the lowest priority and 9 as the highest. According to the JMS specification, clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority. JMS does not require that a provider strictly implement priority ordering of messages (i.e., messages of higher priority may jump ahead of previous lower-priority messages). However, a JMS provider should do its best to deliver expedited messages ahead of normal messages.

In OppNets, there is no guarantee of end-to-end connectivity between the source and the destination and there is no central unit which controls all the network. In the ab-

```
———————— Header ————————

JMSMessageID= ID:FF789
JMSDestination= (Topic) ChatRoom1
JMSTimeStamp= 15-10-11 20:54:03
JMSExpiration= 15-10-18 20:54:03
JMSDeliveryMode= PERSISTENT
JMSPriority= 4

———————— Properties ————————

language= English
locked= true

———————— Body ————————

"Hello World!..."
```

(a) JMS Message

```
———————— Descriptor ————————

id= FF789
destination_id= ChatRoom1
destination_type= topic
date= 15-10-11 20:54:03
deadline= 15-10-18 20:54:03
delivery_mode= PERSISTENT
priority= 4
language= English
locked= true

———————— Payload ————————

"Hello World!..."
```

(b) JOMS Message

Figure 4.5: Example of a JMS message and its JOMS mapping

sence of these conditions, guaranteed message delivery along with expedited priority as defined in the JMS original specification can hardly be guaranteed in an OppNet (unless specific assumptions are made on the characteristics of this network).

JOMS however uses these JMS properties to define a metric, that combines the requested delivery mode and priority level. This metric is defined in a *privilege()* function, which can be redefined if needed by an application developer. It is used to increase the delivery probability for "important" messages by adjusting the cache management policy and the message selection policy of the communication layer. By default, this function privileges *persistent* messages in preference to *non-persistent* ones. The value of message priority (between 0 and 9) is then considered to determine the message's privilege level. The higher the number, the more privileged the message is. Stated another way, the privilege level as stated by the *privilege()* function ranges from non-persistent messages with a priority 0 (the lowest privilege level) to persistent messages with a priority 9 (the highest privilege level). When a new message is created or received while the local cache exceeds its capacity, messages with lower privilege level are discarded first. When a contact is established between two devices, messages with higher privilege level are transferred first.

An example of a JMS message is shown in Figure 4.5a. This message has standard header fields: *JMSDestination* and *JMSMessageID* are used to identify and route the message; *JMSTimestamp* and *JMSExpiration* give the times the message was sent and will be out-of-date. Furthermore, *JMSDeliveryMode* and *JMSPriority* are intended for message control purposes. Two extra properties, *language* and *locked*, have been added for describing the message content. Finally, the message *body* is composed of a short text.

Since the actual implementation of JOMS is based on DoDWAN, it adopts its message model by mapping the JMS message's fields to the DoDWAN message. The JMS message's header and properties are mapped to the DoDWAN message's descriptor, as their content is needed by JOMS to process the messages delivery. Furthermore, the JMS message's body is carried in a DoDWAN message as its *payload*, and considered as a byte

```
destination_id= "ChatRoom1"
language= "English|Chinese"
```

(a)

```
destination_id= "ChatRoom2"
language= "French"
```

(b)

Figure 4.6: Example of selection properties

array. The JOMS message of the already-presented JMS message is shown in Figure 4.5b.

4.5.2.2 Publish-Subscribe Model

JMS providers usually implement this communication pattern using a server-based model: publications and subscriptions to a given topic are managed by a central entity. Since this centralized approach is hardly applicable in OppNets, the principles of content-based networking are used to achieve the same semantic in such environments.

JOMS actually leverages its communication API so as to provide the JMS publish-subscribe model. Messages published in a given topic are tagged with the topic name. The message shown in Figure 4.5b represents an example of a message published to the topic "ChatRoom1".

All subscribers to a topic define a pattern with this topic's name in their interest profile. Thus, a message for a given topic disseminates in the network thanks to the topic's subscribers, that serve as data mules and carry it in their local cache. Topic subscribers can filter out messages beyond the topic name, though, using a selector property as defined in the JMS specification. This property is added to the interest profile, so message filtering is processed by the opportunistic content-based communication layer. Two selection properties are shown in Figure 4.6. Each selection property is composed of the name of a destination object (the first line) and a selector property (the second line). A subscriber with the selection property shown in Figure 4.6a will receive a message such as shown in Figure 4.5b for its profile matches this message's descriptor. In contrast, another subscriber with the selector shown in Figure 4.6b will not.

According to the JMS specification, a subscription to a topic can either be *durable* or *non-durable*. On the one hand, a *non-durable* subscriber can receive only messages that are published while it is active. When a non-durable subscriber becomes inactive, it is considered to no longer exist and it will not receive any subsequently generated messages published to the topic. On the other hand, a durable subscriber continues to receive messages, whether active or inactive, by storing the non-delivered messages until being consumed by the durable subscriber. In OppNets, mobile devices are switched off frequently in order to preserve battery budget. As a result, the implementation of the JMS *non-durable* subscriptions concept, where messages are delivered only to always-on devices (i.e., active subscribers using JMS terminology), is unsuitable and has no meaning for these environments. This issue is addressed in the actual implementation by introducing a way to configure the behavior of JOMS regarding non-durable subscriptions. By setting or unsetting the corresponding property, JOMS automatically considers all non-durable subscriptions as durable ones, or simply refuses non-durable subscriptions and reports any attempts to use them by throwing exceptions.

4.5.2.3 Point-to-Point Model

This model is built around the concept of queue, which has a central role in this communication model: a message sent to a queue must be consumed by only one queue receiver.

In fixed platforms, queues are maintained on a server, which plays this central role in selecting one receiver among multiple potential recipients associated with the queue.

In an OppNet environment, achieving the semantic of JMS queues, where a server-based implementation is inappropriate, could be done using one of the following solutions:

- Multi-cast solution: this solution achieves the semantic of sending a message to a queue by multi-casting this message to the potential receivers associated with the queue. The receivers must leverage a sort of consensus algorithm so as to elect one receiver who can consume this message.
- Global list solution: each queue sender keeps a list of the receivers associated with each queue. Hence, it is up to the sender to choose one receiver from the list so as to unicast messages to this receiver. The problem here is that, for each queue, senders must keep the corresponding list up-to-date. Another problem stems from the fact that such a technique requires each sender, when participating in many point-to-point sessions, to reserve resources for each queue.
- Any-cast solution: in this solution, only one copy of each message circulates in the network. This message will be then consumed by the nearest receiver. This solution decreases noticeably the message delivery ratio in an OppNet since there is no duplicate messages to be used if the original one is lost. Furthermore, the potential receivers which are far from senders may never get a chance to receive any message. As a result, implementing such solution will not be effective in OppNets.
- Quasi-central queue: according to the JMS specification and suggested design patterns, it is common and preferable for a JMS client to have all of its messages delivered to a single queue [81]. In order to enforce this rule in OppNets, each queue can be maintained by a single queue manager (*QM*), which is the device on which the queue is created. Messages sent to a queue, and requests to get messages from that queue, are thus forwarded towards the corresponding *QM*. In an OppNet, each sender/receiver associated with a given queue serves as a mobile carrier for messages related to this queue. To this end, even if a *QM* is turned off or becomes unreachable, it receives later all the missing requests and messages cached on the mobile carriers. Thus, a *QM* principally acts as a *central decision-maker*. JOMS adapts this solution in order to achieve the semantic of JMS queues in OppNets. Note that when several requests reach the same *QM* simultaneously, the *QM* ensures that each message read from the queue is sent to only one receiver, thus enforcing the above mentioned rule. Additionally, requests can include a selection pattern, so the *QM* searches in the queue for a message whose properties match this pattern. A *QM* with the selection pattern shown in Figure 4.7a will receive all the requests and messages for the queue "Alan@00b0d086bbf7". A JMS client can act as a receiver for this queue by sending the request shown in Figure 4.7b. By doing so, the client

```
destination_id= "Alan@00b0d086bbf7"
destination_type= "queue"
```

(a) A queue manager's profile

```
destination_id= "Alan@00b0d086bbf7"
destination_type= "queue"
operation_type= "request"
src= "86f8f700dad0"
language= "English"
```

(b) A queue receiver's request

Figure 4.7: Examples of message descriptors and message selectors

"*86f8f700dad0*" requests the queue "Alan@00b0d086bbf7" to be a receiver for English messages. When an English message is received by the *QM*, it is forwarded immediately to that JMS client. Whenever several receivers are interested in the same kind of messages (e.g., English messages), the *QM* applies a selection policy so as to choose one receiver in a fair way.

4.6 Deploying an Already-existing JMS Application over JOMS

An obvious advantage of implementing a provider that conforms to the JMS standard specification is that developers do not need to learn a new programming language, or get familiar with an exotic programming model or API. A developer can simply focus on writing a JMS application "as usual", and JOMS can take care of its execution in unusual conditions. Indeed, any pre-existing JMS application can theoretically be deployed using JOMS, and run satisfactorily in an OppNet. As a proof-of-concept, the simple JMS client application tested using the OpenJMS implementation and presented in Code. 4.1, has been deployed successfully over JOMS without changing any line in its source code.

Assuming that the JOMS package is installed in the home directory, this application can be compiled in the following way:

```
% javac -cp $HOME/JOMS.jar HelloWorldMessage.java
```

Once the JMS client has been successfully compiled, it can be run on the sender-side terminal in the following manner:

```
% java -cp $HOME/JOMS.jar HelloWorldMessage sender
```

Similarly on the receiver-side terminal:

```
% java -cp $HOME/JOMS.jar HelloWorldMessage receiver
```

Yet there are a few common pitfalls developers should pay attention to while designing their code. More details about these pitfalls are discussed in Section 4.9.

4.7 Technical Background About OppNets

Before starting to evaluate the performance of JOMS in an OppNet, one must first create such a network between mobile devices. Wi-Fi systems in mobile devices generally support two modes of operations: the so-called ad hoc mode, and the so-called managed mode.

Both modes have been defined in Wi-Fi systems since the first version of the IEEE 802.11 standard, in 1997. The ad hoc mode clearly presents some advantages over managed mode (a.k.a. infrastructure mode), especially because it is a fundamental enabler for creating spontaneous non-administered networks. In the literature, researchers address many issues related to ad hoc networks (e.g., multi-hop routing, power consumption, etc.), assuming that creating such ad hoc networks is a trivial task. However, when developers try to create ad hoc networks between smartphones in real life using the ad hoc mode, they face the following challenge: the ad hoc mode is not supported in stock smartphones.

Indeed, the operating systems of smartphones support ad hoc mode natively, but their network managers do not provide any way for a user to create an ad hoc network. This limitation can be by-passed by having a root access on the smartphones.

The recent Wi-Fi Direct standard, which is sometimes presented as an alternative to the ad hoc mode, is actually a very poor substitute for real ad hoc networking, as it does not support large-scale ad hoc networking.

Indeed, Wi-Fi Direct and the ad hoc mode solve very different use-cases and one cannot replace the other. Wi-Fi Direct is not a real peer-to-peer protocol. It is essentially a protocol for forming hierarchical groups, and mostly used to connect adjacent smartphones in small-scale networks. The Wi-Fi Direct enabled smartphones are only equal peers until they connect to each other, and then one of them assumes the traditional role of Access Point (AP), a.k.a, Group Owner (GO), while the others connect to the GO as simple clients in station mode. The ad hoc mode is a peer-to-peer solution with no hierarchies. It opens up a wide range of communication modes, among them opportunistic and delay tolerant networking over large-scale ad hoc networks.

4.8 Experimental Evaluation

Evaluating the performance of a middleware system capable of running in an OppNet is a challenge. In the literature, protocols and systems designed for OppNets are often evaluated through simulation, and little or no effort is devoted to producing code, that can be used in a real setting.

A salient feature of JOMS is that it has been fully implemented, and is now distributed under the terms of the GNU General Public License[5]. Moreover its effectiveness and efficiency have been evaluated in real conditions. To this end, several measurement campaigns have been conducted, using either smartphones or netbooks as host devices. During these experiments it would have been most interesting to compare JOMS against other JMS providers. Unfortunately, JOMS is currently the only JMS provider for OppNets that is openly available for application developers. It can thus be tested in real conditions.

As a general rule any author working on opportunistic networking or opportunistic computing faces the same issue: evaluation methods that are commonly used in stable

[5] `http://www-casa.irisa.fr/joms`

connected networks are inapplicable in OppNets. In the literature JMS providers are usually evaluated by measuring how fast distributed processes can access remote message queues and topics. Since most of these providers are meant to run in connected networks, the results are often presented in terms of throughput and overhead, assuming the underlying network is quite stable and assuming its raw capacity is known in advance. As explained in Section 2.2.3 a middleware system designed for OppNets can do no magic: unless otherwise specified its does not control how mobile hosts move in the network, so it cannot guarantee that a message will ever reach (or reach in time) any particular destination. The behavior JOMS can show in an OppNet is therefore highly dependent on how this network evolves over time. Measuring the throughput of message transmissions between two mobile hosts, for example, does not make much sense if no transmission path ever exists between these hosts.

This section describes in detail the experimental setup used to evaluate the effectiveness and the efficiency of JOMS in providing JMS services in an opportunistic environment. It also presents the results obtained through this experimental analysis.

4.8.1 Resource Consumption

Mobile devices are generally characterized by limited processing power, memory size, and power budget. In order to determine how much resources JOMS consumes once deployed on smartphones, a series of tests were performed on an Android service that have been developed using JOMS.

The evaluation was performed in an indoor office environment using identical HTC Wildfire S smartphones that run Android 2.3.5 operating system. These smartphones have a 600 MHz Qualcomm MSM7227 processor, a usable RAM of 418 MB and a Lithium-ion battery with capacity 1230 mAh.

In the first test, JOMS was kept running as a background Android service, with no application-level traffic but the episodic gossiping used by JOMS for neighbor detection and coordination[6]. During this test which lasted for 14.5 hours before the battery was drained, Android services together consumed 9.25 minutes of CPU activity (about 1% duty cycle), while JOMS itself consumed 16 seconds of CPU activity (about 0.03% duty cycle). Besides, once installed JOMS consumed about 3.63 MB of memory (with an empty message cache), which represents less than 1% of the 418 MB available on a Wildfire S smartphone.

In a second series of tests, the effect of running JOMS on the battery life of a smartphone was measured. Ideally battery consumption should be expressed in *mAh*, but unfortunately the Android system only triggers an event when the remaining battery life decreases by 1%, as estimated by the built-in battery controller. Since this controller uses its own (non-documented) heuristics to estimate the battery's lifetime (based on the age of the battery, on charge/discharge history, etc.) the effective consumption in *mAh* can hardly be obtained by multiplying the estimated lifetime (in %) by the theoretical maximal capacity of the battery. The most basic approach to measure accurately the battery drain would be to insert a multimeter between the battery and the smartphone. This technique has been reported in the literature [82, 83, 84]. However, it is beyond the scope

[6]Measurement taken with a third-party application (System Tuner).

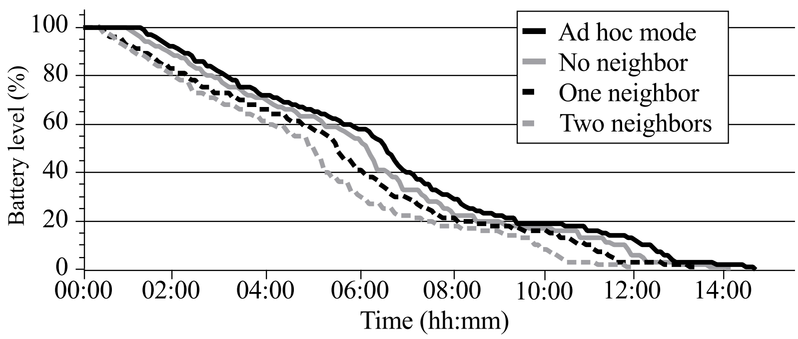


Figure 4.8: Battery drain when running JOMS in the background on a smartphone (in ad hoc mode)

of this thesis, especially since the objective is simply to observe power consumption at a rather coarse grain.

As a baseline, the battery drain on a smartphone is measured running no application, with the Wi-Fi chipset enabled and operating in ad hoc mode. The test is then repeated several times with JOMS running and sending a new message every 3 minutes, with a varying number of neighbors. The evolution of the battery level during these tests is shown in Figure 4.8. It can be observed that the beaconing used by JOMS for neighbor discovery (when no neighbor is there to respond) reduces the battery life of a smartphone by about 5%, and that the gossiping with one neighbor reduces the battery life by an additional 15%. The battery drain does not grow significantly with a higher number of neighbors, though.

This results confirm that the battery drain in a smartphone is mainly driven by the ad hoc mode, and only marginally by the beaconing and gossiping performed by JOMS. JOMS therefore allows a smartphone to run for a long time, without consuming too much resources while running as a background service.

4.8.2 Overhead Assessment of Multilayer Architecture

Since JOMS is implemented on top of an opportunistic communication system, which itself relies on UDP transmissions, the second objective was to evaluate how this multilayer middleware architecture performs over the underlying wireless transmission medium.

During this measurement campaign, two netbooks A and B, running JOMS over a Linux operating system, were used. These netbooks were installed next to each other in the same room, and their built-in Wi-Fi 802.11bg chipsets were configured to operate in ad hoc mode using 802.11g OFDM modulation. Since these hosts were very close to each other the wireless link between them offers a theoretical maximum speed of 54 Mbps. In this scenario the time required to transmit messages of different sizes from one netbook to the other was measured. These messages were produced by host A and received by host B, using either a topic-based model or a queue-based model. In the first case the messages were published to a topic by host A, and B was a subscriber to this topic. In the second case A hosted a local message queue, and B was a subscriber for messages

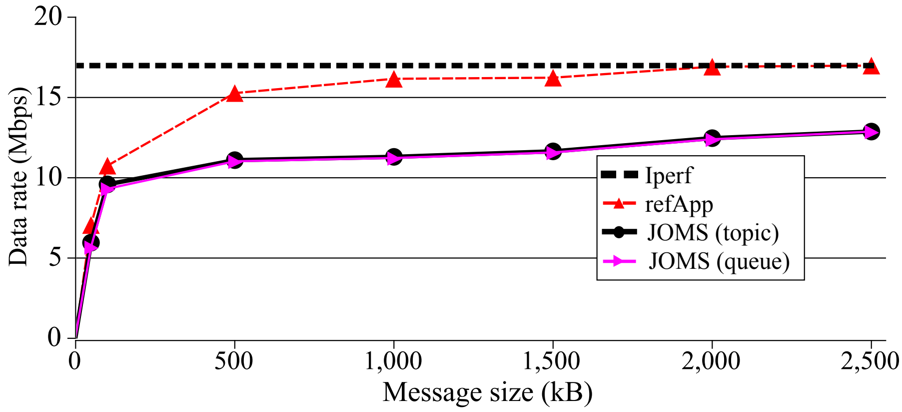


Figure 4.9: Transmission throughputs observed between two directly connected neighbors

deposited in this queue.

In order to get baseline values, the network performance measurement tool *Iperf* is used. This tool permits to measure the maximal throughput achievable on statically defined routing paths at IP level. Furthermore, a simple reference application (hereafter referred to as *refApp*) is developed. It is capable of sending, relaying and receiving data bundles over UDP, with data fragmentation and reassembly if needed. This application permits to measure the throughput achievable at message level, without paying the cost of neighbor discovery or gossiping among neighbors.

To compare with the baseline, different size messages were transmitted, first using the baseline applications (i.e., Iperf and refApp), and second using JOMS. Averaged over 150 rounds, the results of these tests are shown in Figure 4.9, in terms of application-level data rates. It can be observed that topic-based and queue-based messaging obviously show similar performances: the curves obtained with queue-based and topic-based transmissions are actually superimposed in the figure. Furthermore, both of them show between 15% and 20% overhead over *refApp* for large messages.

The delay to produce a message during the previous test is also measured. To do that, we calculated the time elapsed between two moments: the moment when a message is started to be sent at the application layer, and the moment when the message is actually sent at the physical layer. We have measured an average latency of 10 ms with *refApp*, and 25 ms with JOMS.

Considering that JOMS is based on a sophisticated opportunistic protocol, which orchestrates communications between neighbor hosts, it can be considered that the aforementioned results are quite reasonable. Yet JOMS's most salient feature is of course that it can ensure the delivery of messages in an OppNet, where traditional JMS providers (designed for connected environments) as well as that simple applications (e.g., *refApp*) are totally useless.

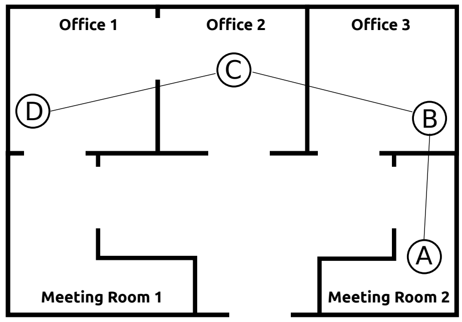


Figure 4.10: Multi-hop forwarding test scenario

4.8.3 Efficiency of JOMS over a Single Connected Island

Before trying to observe how messages can propagate network-wide in an OppNet, we strove to observe how they can propagate among connected devices, that is, within a single connectivity island.

To do so we used four netbooks deployed in adjacent rooms so as to form a straight line A-B-C-D, as shown in Figure 4.10. Because of the effect of concrete walls on signal attenuation, the connectivity links between these netbooks were such that each netbook could only communicate with up to two neighbors (e.g., B could only reach A and C). The tests relied on the queue-based transmission model. Message queues were created on B, C, D, and netbook A was configured as a subscriber to all these queues. Messages with sizes varying between 1 kB and 2.5 MB were deposited successively in each queue (by local deposits), and the time required for these messages to reach subscriber A is measured .

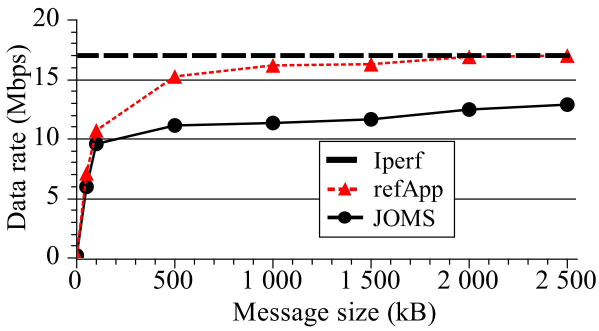
In order to get baseline values, the network performance measurement tool *Iperf* along with the aforementioned *refApp* application are used. Of course *Iperf* and *refApp* can both only be used in a connected island, as they require temporaneous end-to-end connectivity and multi-hop forwarding.

The results of the tests are shown in Figure 4.11a, 4.11b and 4.11c for transmissions over 1, 2 and 3 hops, respectively. Each value presented has been averaged over 150 rounds, for each configuration.

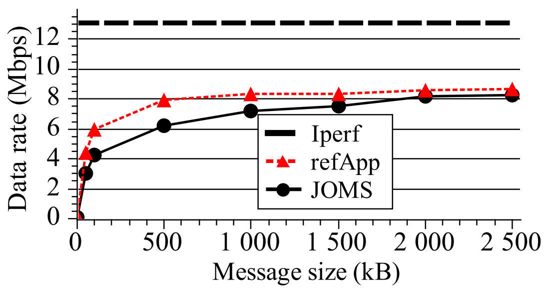
It can be observed that JOMS shows 5% to 20% overhead over *refApp*, which is an indication of the cost of discovering neighbors and gossiping with them before exchanging actual applicative messages. This overhead gets lower when multi-hop forwarding is required, for in that case *refApp* must receive messages before forwarding them, just like JOMS. Moreover, the observed transmission rates globally decrease as the number of hops increases. This is because when host B must serve as a relay between A and C, the radio channel around B is twice as busy as when B interacts only with host A. The same observation applies for host C when it must serve as a relay between B and D.

4.8.4 Efficiency of JOMS in a Real OppNet

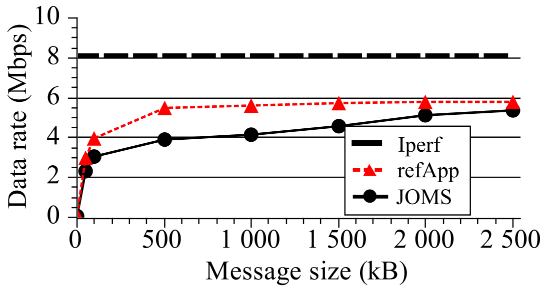
The above-mentioned tests demonstrated that JOMS can perform satisfactorily in a traditional connected environment, and show reasonable performances in such conditions.



(a) Between 1-hop (direct) neighbors



(b) Between 2-hop neighbors



(c) Between 3-hop neighbors

Figure 4.11: Transmission throughputs observed in a single connected island (with or without multi-hop forwarding)

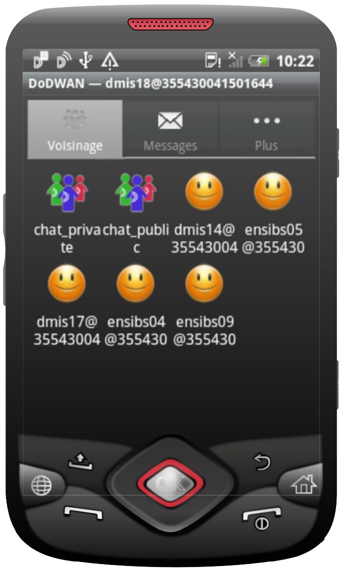


Figure 4.12: A SMS-like Android application offering public text (group icon) and private text (smiley icon)

However, a disruption-tolerant system such as JOMS is not really necessary in a connected environment, as a traditional JMS provider combined with multi-hop forwarding would also run satisfactorily in such conditions. Yet the most interesting characteristic of JOMS is of course that it can ensure the delivery of messages in a disconnected network, where traditional JMS providers –or simple programs like *Iperf* or *refApp* for that matter– would be totally useless.

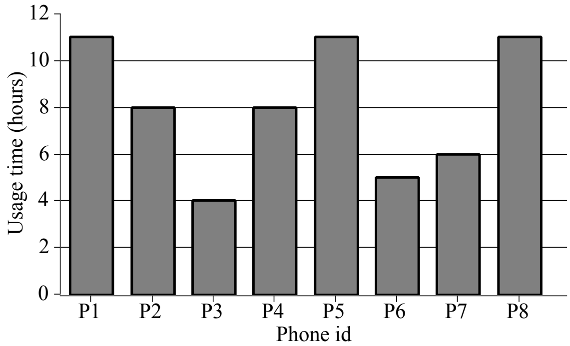
In order to demonstrate that JOMS can indeed support JMS messaging in an OppNet, eight volunteers were equipped with HTC smartphones. These devices ran a simple SMS-like[7] application based on JOMS, as shown in Figure 4.12. This application offers two services: public text and private text. The public text service relies on topics: each message published in a topic can be received by all subscribers. With the private text service, each message is addressed to a single message queue, which is hosted on the smartphone of the recipient user.

This experiment spanned over the working hours of two consecutive days for a total duration of 32 hours. The volunteers were asked to carry their smartphone whenever possible, and use both text services while moving inside IRISA laboratory or in its surroundings. The experiment was mostly meant to serve as a proof-of-concept.

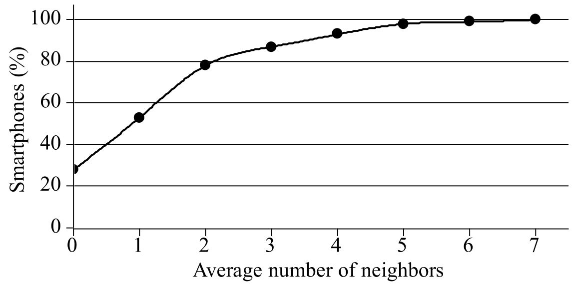
Figure 4.13a shows how long each volunteer actually used his/her smartphone (i.e., with the screen lit) during the experiment. It can be observed that the smartphones remained untouched on average during three-fourth of the experiment duration. This does not imply that JOMS remained idle during that time, though, as it can keep gossiping with neighbor devices when running in the background.

The cumulative distribution function (CDF) of the average number of neighbors perceived by all smartphones is shown in Figure 4.13b. It can be observed that, in average, a smartphone did not have more than two neighbors simultaneously during almost 80% of

[7]SMS: Short Message Service.



(a) Usage duration of smartphones



(b) Cumulative distribution of the average number of neighbors perceived by all smartphones

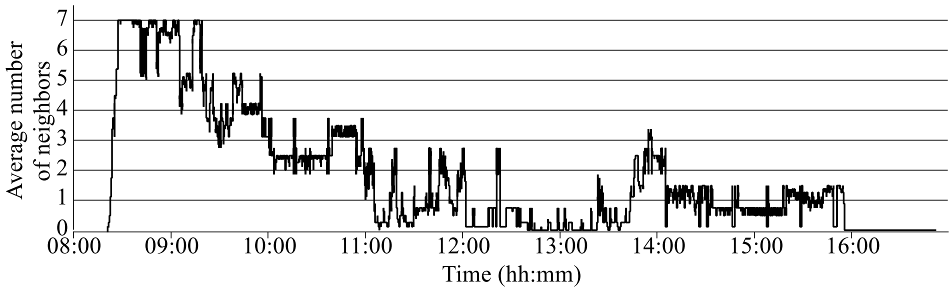
Figure 4.13: Activity and "sociability" of smartphones

the duration of the experiment, and that it was actually alone (i.e., with no neighbor) during about 30% of that time. These figures confirm the disconnected nature of the network formed by the smartphones, and demonstrate the need for opportunistic transmissions in order to maintain communication in such conditions.

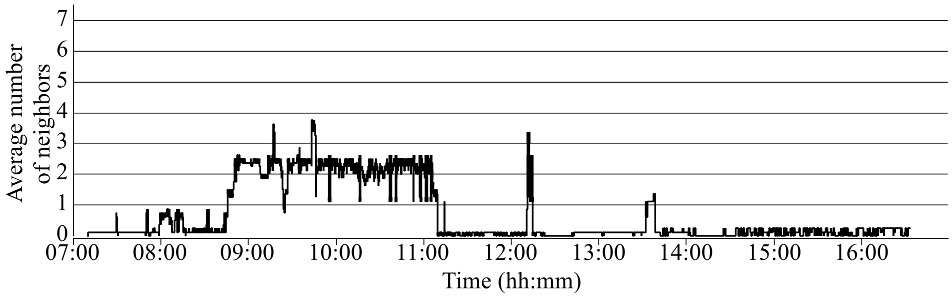
A timeline of the average number of neighbors during the campaign is shown in Figure 4.14. The high number of neighbors during the first two hours of the experiment is explained by the fact that all volunteers were in the same room at that time, as the purpose and conditions of the experiment were explained to them. Later on all volunteers dispersed in the laboratory, and the timeline only shows episodic contacts between their smartphones. These results again confirm that a communication system requiring end-to-end connectivity, and global access to any kind of server, would be totally impractical in such conditions.

The traces collected by JOMS on each smartphone show that 230 text messages were sent by the volunteers during the two-day experiment. Figure 4.15 presents the cumulative distributed function (CDF) of the delivery time for topic-based and queue-based messages. It can be observed that nearly 80% of topic-based messages got delivered in less than 15 minutes, whereas most queue-based messages took about 40 minutes to be delivered. This difference is due to the fact that a message for a queue is destined to one receiver. In contrast, a message for a topic is destined to several subscribers so it has more opportunities to get delivered by some subscribers before that a queue-based message gets delivered by its final receiver. In any case, the figures show that during the experiment most of the messages got delivered to their destination(s) in less than 1 hour. Yet, about 3% of the messages could not be delivered. This is the consequence of the unpredictable –yet perfectly legitimate– behavior of the volunteers, who sometimes moved away from the campus, or even switched their smartphone off in order to preserve its battery budget. By doing so they prevented any further radio contact between their smartphone and those of other users, and this of course led to message loss.

During the experiment, 5729 contacts were observed between smartphones, with an average contact duration of 121 seconds. The distribution of contact durations is presented in Figure 4.16. The majority of radio contacts lasted for less than a minute. Furthermore, 2.5% of radio contacts were not exploitable by JOMS because they were asym-



(a) First day



(b) Second day

Figure 4.14: Timeline of the average number of neighbors during the last campaign

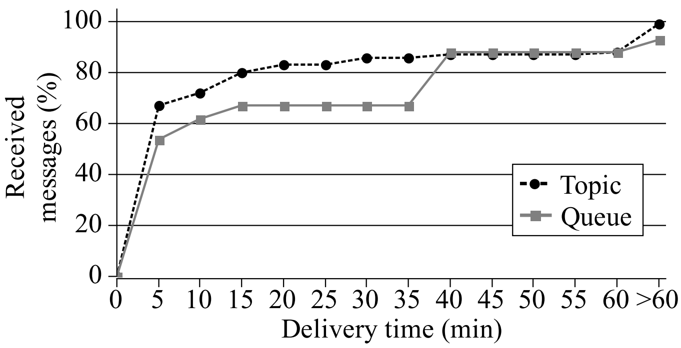


Figure 4.15: Cumulative distribution function of the delivery time of received messages

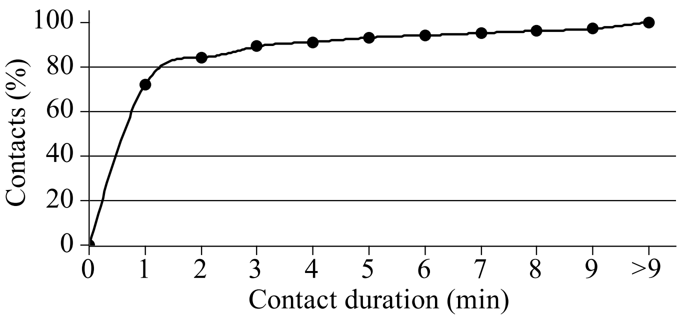


Figure 4.16: Cumulative distribution function of contact durations

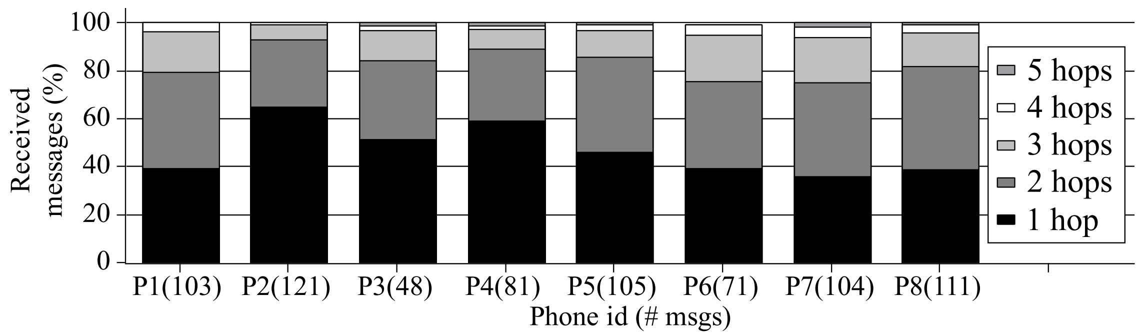


Figure 4.17: Amount of messages received in a direct/multi-hop manner by each smartphone

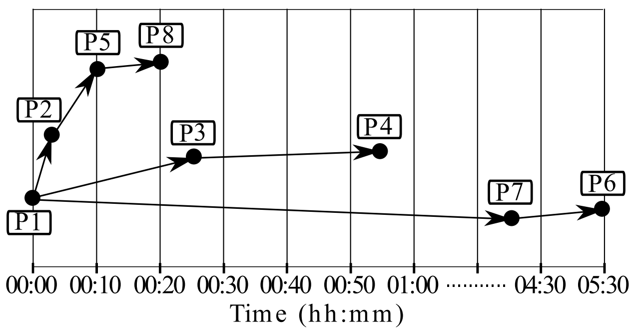


Figure 4.18: Timeline of the dissemination of a message

metric, that is, one smartphone could detect a neighbor, but the reverse was not true. Such a situation usually happens for only a few seconds, and it occurs when the smartphones involved are too far from each other, so their signal is barely above noise level. The obtained results confirm the transience of wireless links in an OppNet composed of smartphones moving unpredictably. Systems, that must rely on transient contacts to transport data between smartphones must definitely resort to disruption-tolerance communication system.

The extent of multi-hop forwarding is another important metric, which has been monitored during this campaign. Figure 4.17 shows the number of messages received by each smartphone (in parentheses), and the distribution of the number of hops required for messages to reach that smartphone. It can be observed that most of the smartphones received at least half of the messages after a multi-hop trip. This observation confirms the interest of having smartphones serve as benevolent message carriers in an OppNet such as that considered in this experiment. In order to better illustrate this point, Figure 4.18 shows how one particular message disseminated during the experiment. This message was first published in the public topic by mobile *P1*. After only a few minutes *P1* established a radio contact with *P2*, which thus got a copy of the message and became a new carrier for this message. *P1* later managed to forward the message to *P3*, and later to *P7*, while *P2* forwarded it to *P5*. The message thus kept disseminating, until it reached the last subscriber *P6*, about 5 hours and 30 minutes after it was initially published.

4.9 Discussion

The above-mentioned results demonstrate that JOMS is effective and efficient in providing JMS services in OppNets. This section covers the side effects of implementing a standard specification, originally designed for "traditional" networks, in OppNets, knowing that such technique, if not used carefully, can have both favorable and unfavorable consequences.

On the one hand, developers do not need to learn a new programming language, or get familiar with an exotic programming model or API. A developer can simply focus on writing a JMS application "as usual", and JOMS can take care of its execution in unusual conditions. Indeed, any pre-existing JMS application can be deployed over an OppNet using JOMS. As a proof-of-concept, Section 4.6 showed how to deploy a simple JMS client application successfully over JOMS without changing any line in its source code.

On the other hand, developers should tolerate the specific constraints posed by OppNets, where message ordering, message delivery and transmission delays are usually not guaranteed. It is important to understand that such constraints are not due to limitations in JOMS: they are due to the very nature of OppNets. As explained in Section 2.2 opportunistic protocols and middleware systems designed for OppNets can do no magic: they strive to support network-wide communication in an OppNet, using mobile hosts as carriers, that help bridge the gap between non-connected parts of the network. Yet, unless otherwise specified they do not control how mobile hosts move in the network, so they cannot guarantee that a message will ever reach (or reach in time) any particular host in the network. A developer working on an application for OppNets should therefore recognize that unordered delivery, delivery failures and late deliveries may be more common than ordered/in-time deliveries, and design the application or organize its deployment accordingly, as explained in Section 2.3.2.

Yet there are a few common pitfalls developers should pay attention to while designing their code. Indeed, many standard applications based on JMS have been built with the implicit –but often wrong– assumption that the JMS provider can guarantee message ordering. According to JMS specification, message order is the order of messages sent by a session which it is not necessarily the same order as produced by clients. Furthermore, the specification does not define an order of messages receipt across multiple destinations or multiple sessions. In such cases, message order will not be under provider control. Yet, the message order supported by all current JMS implementations is provided thanks to the flow control supported by the protocol TCP. It is worth noting that the semantic of message order obtained using the flow control of TCP is totally different from the semantic defined in JMS specification. One of the best ways to avoid such kind of problems is in fact to avoid developing with preconceived ideas and read carefully the specification in order to design applications accordingly.

4.10 Related Work

Opportunistic computing is actually becoming a new distributed computing paradigm, involving transient and unplanned interactions between mobile devices [17]. Some middleware systems have been proposed in order to ease the task of application programmers for mobile ad hoc networks (MANETs) [85], providing well-known higher-level programming models or designing new ones. The approach, presented in this chapter, and those discussed below clearly fit in the first category, as they consist in developing JMS providers in order to provide publish-subscribe programming model and message-queues programming model for ad hoc networks.

Vollset et al. describe a JMS implementation supporting non-persistent topics in MANETs [86]. This implementation uses a multicast routing protocol to provide publish/subscribe semantics by mapping JMS topics to multicast addresses. Since this protocol cannot disseminate messages beyond a single connected fragment of the network, it is hardly usable in OppNets. It could probably be adapted, though, using a disruption-tolerant multicast routing protocol. In this system, no directory service (such as JNDI) is provided: all JMS clients are assumed to own a local copy of a configuration that contains information about all queues and topics. In the paper the authors propose some solutions to support JMS queues, but none of these solutions has actually been implemented.

JIMS (JMS Implementation for MANETs using Sociable nodes) has been designed during a Master's project at University College London [87]. It defines the notion of *social nodes*. A social node is one that is supposed to meet many other nodes while moving in a MANET. It also has a lot of resources, so it can carry many messages, and never needs to be turned off while roaming the network. This idea of having privileged nodes that can serve as effective data mules is an interesting one. A similar behavior can actually be obtained with JOMS, as the amount of resources used by the system on each device can be adjusted accurately. JIMS guarantees the one-to-one communication model by maintaining only one copy of each message in the network. JIMS does not support a distributed directory service; instead, each node has it is own set of configuration files, which store locally-created administered objects.

Epidemic Messaging Middleware for Ad hoc networks (EMMA) is an adaptation of JMS, that targets MANETs presenting connectivity disruptions [81]. EMMA assumes the availability of a so-called *synchronous protocol*, which can be used to reach mobile hosts that belong to the same *cloud* –or island– as the sender. An asynchronous epidemic routing protocol is used to disseminate messages towards remote clouds. EMMA manages queues in a manner that is quite similar to that of JOMS: each queue is maintained by a single holder, which advertises this object periodically with a set lifetime, and which can accept subscriptions from other hosts. EMMA and JOMS differ in the way they deal with topics. In EMMA topics are maintained by a single holder, which can accept subscriptions from other hosts. In JOMS topic subscriptions can be set locally on any host. Another difference is that in EMMA the gossiping mechanism between neighbor hosts is such that all messages are systematically considered, so very large lists of message identifiers can be exchanged between neighbor hosts. In JOMS this gossiping is constrained by the interest profiles of neighbors. Finally, EMMA defines its own communication protocol for route discovery and maintenance, while JOMS presents a two-layer architecture: the upper layer is concerned with queue and topic management and utilization, and the lower layer supports opportunistic communication.

Although [86], [87] and [81] have been published almost a decade ago, the JMS providers they describe have never been openly distributed. JOMS is currently the only provider that is available for developers that wish to develop JMS-based distributed applications for OppNets.

Table 4.1: Comparison between the outlined JMS providers

JMS Provider	Disruption tolerant	Message queues programming model	Publish-subscribe programming model	Directory service	Openly distributed
JMS on MANETs [86]	x	x	✓	x	x
JIMS [87]	✓	✓	✓	x	x
EMMA [81]	✓	✓	✓	✓	x
JOMS	✓	✓	✓	✓	✓

Table 4.1 summarizes the comparison between the aforementioned JMS providers.

4.11 Summary

This chapter presented JOMS *(Java Opportunistic Message Service)*, a message-oriented middleware system, that is meant to provide two high-level programming models in OppNets: publish-subscribe programming model and message queues programming model. It is basically a JMS provider, but unlike other providers it does not rely on a central message repository or service directory. Administered objects are distributed in the network, and a distributed directory service is used to discover and locate them. Information sharing relies on a content-driven epidemic scheme, whereas all mobile devices collaborate to disseminate messages network-wide. This combination of unique features makes it possible for JOMS to perform effectively in an OppNet, while other JMS providers would be ineffective in similar conditions.

JOMS has been tested and evaluated in real conditions using Android-based smartphones (as reported in this chapter), but also using tablets, laptops, and netbooks.

The source code of JOMS is openly available for researchers and developers, and provides a readily available platforms to develop full-featured distributed applications, that can benefit users of OppNets, especially in disaster or economically challenged areas.

JMS programming model provides few primitives that are all limited to very specific data structures. In fact, JMS applications can only communicate using JMS messages. Furthermore, developers generally need some programming models, that allow them to design distributed applications, that can collaborate more efficiently in OppNets. In this respect, another programming model is needed in order to support the distributed exchange of generic data structure in OppNets so as to allow distributed applications to coordinate their actions more efficiently in these environments. This is discussed in details in the next chapter.

5

A Tuple Space Implementation for Opportunistic Networks

Contents

5.1 Introduction

THIS chapter introduces JION (*JavaSpaces Implementation for Opportunistic Networks*), a novel JavaSpaces implementation specifically designed for OppNets, that permits the provisioning of the tuple space programming model over these environments. Its major aim is to provide an efficient coordination middleware system to allow entities (i.e., system peers) to coordinate their actions in a disruption-tolerant way.

A main advantage of adapting the standard JavaSpaces specification is to increase portability, as that pre-existing JavaSpaces-based applications shall theoretically be easily deployed and run satisfactorily in an OppNet. Indeed, developers do not need to learn a new programming language, or get familiar with an exotic programming model or API. A developer can simply focus on writing a JavaSpaces application "as usual", and JION can take care of its execution in unusual conditions.

This chapter gives some background about the basic concepts of JavaSpaces and Future objects, along with their importance in the context of OppNets. It then presents

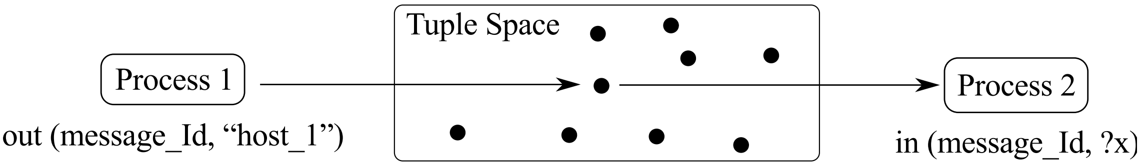


Figure 5.1: A simple one-to-one communication pattern in Linda

JION, a peer-to-peer JavaSpaces implementation specifically designed along this line to provide the tuple space programming model in the context of OppNets. As a proof-of-concept, a real application was developed using JION, which was then used to evaluate the efficacy of JION in real-life measurement campaigns.

5.2 Background

JavaSpaces is a Java specification of the tuple space programming model, which was originally introduced in the Linda programming language [88]. This section provides a brief introduction to this programming model as introduced in Linda, and to the JavaSpaces technology, which provides an implementation for Java applications.

5.2.1 Tuple Space

The tuple space programming model has its root in the Linda parallel programming language developed in the mid-1980's at Yale University [88]. A tuple space is a shared data space acting as an associative memory used by several clients, called *processes*, for communication and/or coordination requirements. A Linda application is viewed as a collection of processes cooperating via the flow of data structures, called *tuples*, through a *tuple space*. Each tuple is a record of typed fields containing the information to be communicated. An example of a tuple with two fields is *(message_Id, "host_1")*, where *"host_1"* could be considered as the identification of the message represented by this tuple.

The coordination primitives provided by Linda allow processes to insert a tuple into the tuple space (*out*) or retrieve tuples from the tuple space, either removing those tuples (*in*) or preserving the tuples in the space (*read*). For retrieving operations, tuples are selected using simple pattern-matching from a given set of parameters. For a sake of illustration, a simple one-to-one communication between two process can be expressed using a combination of *out* and *in* operations, as shown in Figure 5.1. A process can deposit the aforementioned tuple in the tuple space using: *out (message_Id, "host_1")*. Retrieving this tuple can be performed using: *in (message_Id, ?x)*, where *message_Id* is the only defined field which will be used to locate the previously deposited tuple. The other field which is denoted by a leading *?*, is simply considered as a wildcard.

Other forms of communications (e.g., one-to-many broadcast operations, many-to-one aggregation operations, etc.) can be easily synthesized from the basic operations of the Linda model. For example, an one-to-many broadcast operation can be synthesized by an *out* operation followed by several *read* operations.

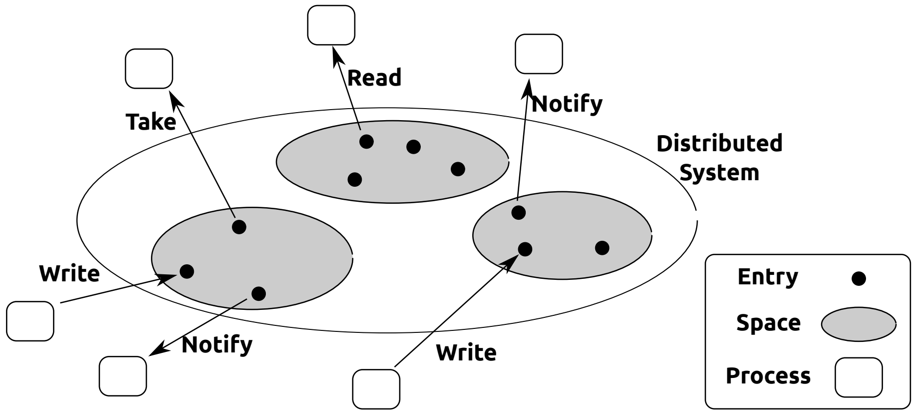


Figure 5.2: JavaSpaces architecture

5.2.2 JavaSpaces

JavaSpaces is a Java specification of the tuple space programming model, introduced as a part of the Java Jini technology. It defines a set of application programming interfaces (APIs) that extend the simple core of Linda primitives. The JavaSpaces version of Linda tuples, called "entries", are Java objects that contain public fields, that act as typed fields in Linda. JavaSpaces also supports *template*s, which are entry objects that have some or all of its fields set to specified values, so as to characterize the kind of entries a process is looking for. That is, a process uses a template to find *matching entries* whose fields match the template. JavaSpaces provides *read*, *take* and *write* operations in order to implement Linda's *read*, *in*, and *out* operations respectively, as shown in Figure 5.2. Additionally, it provides a *notify* operation, that allows processes to be informed of the existence of a matching entry. This operation notifies the processes by sending an *event* containing information, to which the processes can react. It is worth noting that both *read* and *take* operations are blocking operations in which the calling process is suspended while waiting the answer. JavaSpaces also provides *immediate return* versions of *read* and *take* operations, with which processes query a space and immediately return either a matching tuple or a *null* indicating that no matching entry exists. These operations, called *readIfExist* and *takeIfExist*, can be useful when a process requires an immediate answer.

As the entries of JavaSpaces are passive data, processes cannot perform operations on tuples directly. In order to modify an entry, a process must explicitly remove, update and reinsert it into the space.

5.3 Motivation for Providing JavaSpaces for OppNets

An interesting feature in the JavaSpaces programming model is that it provides space and time decoupling. As discussed in Section 2.4.3.1, s*pace decoupling* means that an entry placed in a space may originate from any sender process, and may be delivered to any potential recipient. Furthermore, *time decoupling* means that an entry placed in a space will remain there (potentially indefinitely) until it is removed explicitly, and hence the sender and receiver do not need to overlap in time.

Together, these asynchronicity levels allow JavaSpaces to be fully distributed in space and time. Such an approach is paramount in an OppNet, where the mobile hosts involved in communication disconnect frequently due to their migration or connectivity patterns.

The standard specifications of JavaSpaces represents another advantage that encourages its adoption in the context of OppNets, although it has not initially been designed for such environments.

5.4 Programming Model

This section covers the main mechanisms of creating a space-based application using JavaSpaces. Then it describes how to use these mechanisms in order to write a simple JavaSpaces application.

5.4.1 Basic JavaSpaces Terminology

The actual interface definition for JavaSpaces is short and compact, as can be seen in Code 5.1. Using JavaSpaces terminology, one should be familiar with the following terms:

Code 5.1 The interface of JavaSpaces

```
public interface JavaSpace{
    Lease write(Entry entry, Transaction txn, long lease);
    Entry read(Entry template, Transaction txn, long timeout);
    Entry readIfExists(Entry template, Transaction txn, long timeout);
    Entry take(Entry template, Transaction txn, long timeout);
    Entry takeIfExists(Entry template, Transaction txn, long timeout);
    EventRegistration notify(Entry template, Transaction txn, RemoteEventListener
        listener, long lease, MarshalledObject handback);
}
```

Entry According to the JavaSpaces specification, an entry is an object reference characterized by its "*fields*".

As shown in Code 5.1, every JavaSpaces method takes an entry as a parameter and four JavaSpaces method return an entry.

In the JavaSpaces terminology, entry fields refer only to the public attributes of entry objects. They are meant to characterize an entry, and they are used while performing matching operations when retrieving entries from the space. Templates are entry objects that have some or all of its fields set to specified values, so as to characterize the kind of entries a process is looking for.

Operations Access to entries must be done through a set of basic operations, which are: *write*, *read*, *take, readIfExist*s, *takeIfExists* and *notify*, as shown in Code 5.1.

Method *write* is used to put entries into a space in the first place.

Both *read* and *take* methods provide a way to search a space. The important difference between them is the way by which they deal with matching entries. If a matching entry is found, method *read* retrieves a copy of the matching entry, whilst method *take* removes the matching entry from the space. Both methods are blocking, that is, if no matching entry is available the process performing the operation is suspended until a matching entry becomes available. They return a *null* if the timeout expires.

Both *readIfExist*s and *takeIfExists* methods are very similar to *read* and *take* methods, respectively. They have exactly the same parameters and return type. However, they differ in their treatment of the timeout value. If no matching entry is found, they immediately return rather than waiting for a matching entry. The timeout parameter for both *readIfExist*s and *takeIfExists* methods specifies how long they should wait for an unfinished *transaction* to complete.

Method *notify* provides a means for a process to be informed when interesting entries are written into a space. A process can register for receiving an *event* when matching entries arrive in the space. In the mean time, the process can proceed with other tasks rather than waiting for the event to occur.

Leasing L*eases* are meant to control the lifetime of entries in a space. When the time for a lease has expired, the corresponding entry is removed from the space.

Transactions They provide a mechanism to deal with partial failures. Processes can perform operations under a transaction, which can complete in two ways: either all operations done with this transaction will succeed, or they will all fail.

5.4.2 Programming with JavaSpaces

Code 5.2 Example of an entry

```
public class Seat implements Entry{
    public String departure;
    public String arrival;
    public Date date;
    public Seat(){};
    public Seat(String departure, String arrival, Date date){
        this.departure=departure;
        this.arrival= arrival;
        this.date= date;
    }
}
```

A simple JavaSpaces application is developed here. This application illustrates the basic principles of JavaSpaces' programming model. This application (namely, the Carpooling application) allows users to share empty seats during a ride with fellow commuters on the same route. For each empty seat, users can use class *Seat* shown in Code 5.2 in order to declare the ride for which the seat is available.

Two seats (also called entries in JavaSpaces terminology) can be created using class *Seat* in the following manner:

```
Seat offer1= new Seat("Paris", "Berlin", new Date(2015,10,22));
Seat offer2= new Seat("Paris", "Madrid", new Date(2015,12,17));
```

These entries are characterized by three fields: *departure*, *arrival* and *date*; which are to be considered during matching operations.

These entries can be put into a space by calling *write* method as shown below:

```
space.write(offer1, null, Lease.FOREVER)
space.write(offer2, null, 3600000)
```

As it is shown the entry *offer1* will never expire, whilst the entry *offer*2 will expire after one hour.

Three requests (or templates in JavaSpaces terminology) are created as shown below:

```
Seat request1= new Seat("Paris","Berlin", null);
Seat request2= new Seat("Paris","Zurich", new Date(2015,12,10));
Entry request3= new Ticket(); // Ticket is a class that implements Entry
```

The template *request1* matches the entry *offer1* for the following reasons: first, the fields *departure* and *arrival* of the template match exactly *by the value* the same fields of the entry. Second, the field *date* with null reference in the template is considered as a wildcard, hence it can match any value in the same field of the entry. Finally, the entry and the template match *by the type* (i.e., both of them have the same type since they are defined using class *Seat*). However, on the one hand, the template *request2* does not match any entry for a *value mismatching* reason. On the other hand, the template *request*3 does not match any entry for a *type mismatching* reason. Indeed, the type of the template *request*3 is neither the same as the entries' nor a super type of the entries'. More details about the matching algorithm can be found in [46].

When a process needs to retrieve the entry *offer1* from the space, it can simply provide the template *request1* as a parameter for the *read* operation in the following manner:

```
Seat answer= space.read(request1, null, Lease.FOREVER)
```

As a result, the process is informed that a ride from "*Paris*" to "*Berlin*" is available on Thursday 22/10/2015.

5.5 Shortcomings of Current JavaSpaces Implementations

Many JavaSpaces implementations have been designed and can now be used for distributed systems running in fixed networks like Apache River[1], Blitz Project[2], GigaSpaces[3], etc.

These implementations do not appear to be suitable for OppNets, as they are all based on the client-server model, which contrasts with the extremely dynamic nature of OppNets. It is necessary to use a peer-to-peer model to support OppNets.

[1] http://river.apache.org
[2] http://www.dancres.org/blitz
[3] http://www.gigaspaces.com

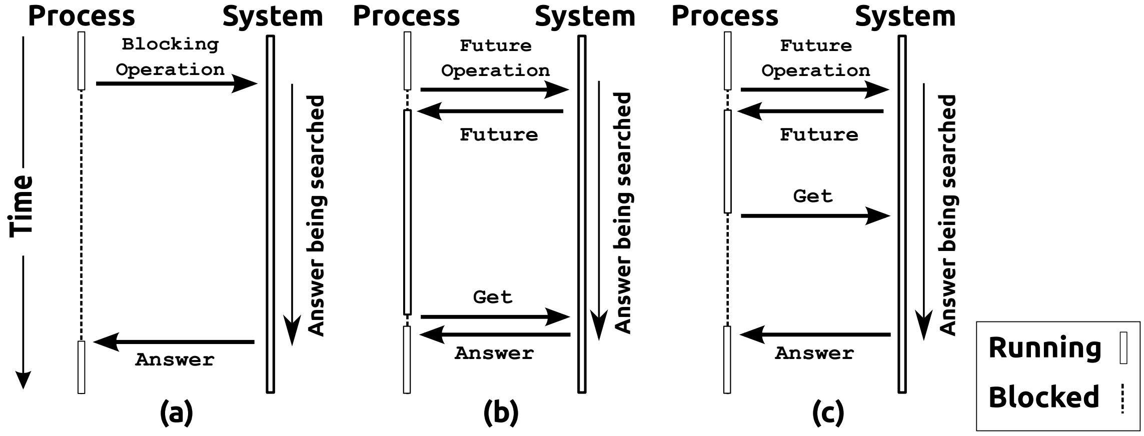


Figure 5.3: Examples of use of Future object

Furthermore, these implementations are based on RMI, that requires the process asking for a service and the server delivering that service to be up and running simultaneously. Indeed, they tolerate neither long transmission delays nor transmission failures. In an OppNet, it is often observed that processes are not connected at the same time, because of deliberate disconnections (e.g., to save battery) or involuntary ones (e.g., no network coverage). In order to cope with this limitation, an asynchronous form of communication is necessary in order to allow interaction between processes that are not directly connected.

In the standard JavaSpaces programming model, blocking operations are natural because unless a process has data to work, with it is suspended. Thus, the *read* and *take* operations are thread-coupling operations (i.e., blocking operations). According to Roman et al., when JavaSpaces systems need to support mobile hosts in ad hoc networks, their operations should not be thread-coupling [89]. Thus, if some needed entry is not available, processes should be able to switch to another task rather than block. Supporting the dynamic nature of ad hoc environments, as reported by Roman et al., requires extending the basic JavaSpaces operations with novel constructs to improve the performance of JavaSpaces and its programming flexibility. Indeed, several JavaSpaces implementations designed specifically for ad hoc networks support thread-decoupling operations. A good survey of these JavaSpaces implementations can notably be found in [90].

Likewise, the JavaSpaces implementation designed along that line extends the standard specification with support for thread-decoupling operations using the technique of Future object [91], as explained in the next section.

5.6 Future Object Background

According to the Future object specification [92], a Future object acts as a *proxy* (placeholder) for the result of not-yet-performed asynchronous computation. A Future object allows a client to continue computation without being blocked, waiting for a not-yet-available answer.

The interface of Future object as defined in Java is shown in Code 5.3. The result of an

Code 5.3 The interface of Future object

```
public interface Future<T>{
    boolean cancel(boolean interruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    T get();
    T get(long timeout, TimeUnit unit);
}
```

asynchronous computation can be retrieved using method *get*, blocking if necessary until the computation has completed and the result is ready. As a consequence, using a Future object can improve computation in such a way that the result acquisition can be achieved concurrently with the calling process, as long as the calling process does not need to invoke methods on the returned object. This trend is illustrated in Figure 5.3.b, which shows the timelines of a process using Future object compared with that without Future object in Figure 5.3.a. However, if the calling process needs to use the returned object, it is then automatically blocked if the result is not yet available, as shown in Figure 5.3.c. Cancellation is possible using method *cancel*, so that the calling process stops waiting for the result, and any resource associated with the Future object can be garbage-collected. Additional methods are provided to determine if the task completed normally or was canceled.

The programming model of JavaSpaces in the presence of mobility, high network dynamism and disconnections can be improved by the provision of thread-decoupling operations, that allow non-blocking access to the space. To do so, the JavaSpaces implementation designed along that line leverages Future object to provide a pair of thread-decoupling operations: *readf* (read in the future) and *takef* (take in the future), as explained in the next section.

5.7 Middleware Architecture and Implementation

The JavaSpaces technology has been primarily designed to provide persistent object exchange areas (spaces), through which processes coordinate actions and exchange data, and this has remained its typical usage scenario. Most JavaSpaces implementations do not appear to be suitable for OppNets. A server-less, flexible and disruption-tolerant JavaSpaces implementation must therefore be developed in order to provide JavaSpaces services for OppNets.

JION (*JavaSpaces Implementation for Opportunistic Networks*) is the JavaSpaces implementation designed along that line. Its general architecture is shown in Figure 5.4. JION is composed of two basic modules: a communication system, and the JavaSpaces system. Currently, JION leverages DoDWAN for communication services, as described in Chapter 3. This section gives more details about the upper layer of JION and its entry model, along with the supported operations.

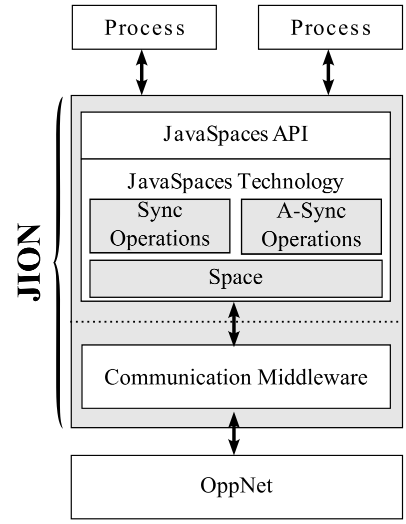


Figure 5.4: JION architecture

5.7.1 System Model

JION is a distributed server-less JavaSpaces implementation, as it is intended to be used in OppNets where server centralization is impractical. Each host maintains a local space, in which JION stores the entries produced locally (that is, entries produced by *write* operations initiated by local processes). Each host is thus responsible for managing its own entries. Indeed, *write* operations in JION are only processed locally, while matching and fetching operations (*read*, *take* and *notify*) are processed by querying hosts over the network for the entries they own.

The *write* operations in JION could however be supported alternatively by propagating the produced entries all over an OppNet and managing them in a collaborative manner. To do so, hosts in the OppNet must leverage a disruption-tolerant consensus implementation so as to coordinate their actions and manage their entries collaboratively. For the sake of illustration, imagine that an entry has been propagated in an OppNet. Each host now has its own copy of the given entry. When a process wants to *take* the entry, a consensus session must be started between the hosts of the OppNet so as to agree to delete the given entry from their cache. Once the consensus is solved, the *take* request can be performed. A disruption-tolerant consensus implementation is discussed in details in the next chapter.

5.7.2 Entries and Templates

As mentioned in Section 5.4.1, an entry is characterized by a set of fields. Templates are special entry objects. The values of their fields are used for matching operation when retrieving entries from a space.

A DoDWAN message has two parts: a descriptor and a payload. Entries and templates are simply carried in the message as its payload, and considered as a simple byte array. Since the descriptor of DoDWAN is meant to characterize the content of the message, JION tags the message to describe its actual content in order to specify if it contains

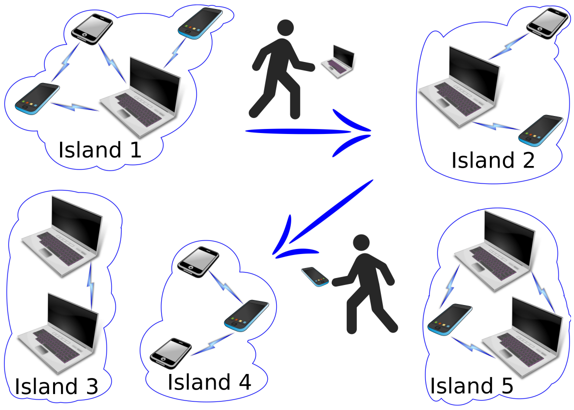


Figure 5.5: Example of an opportunistic network

an entry or a template.

5.7.3 Operations

JION supports the set of operations as defined in the JavaSpaces specification, which are: *write*, *read*, *take*, *readIfExist*s, *takeIfExists* and *notify*. Furthermore, JION leverages Future object to provide a pair of thread-decoupling operations: *readf* and *takef*. The interface of JION is shown in Code 5.4.

Code 5.4 The interface of JION

```
public interface JION extends JavaSpace{
    Future<Entry> readf(Entry tmpl, Transaction txn, long timeout);
    Future<Entry> takef(Entry tmpl, Transaction txn, long timeout);
}
```

Below is a description of the way JION supports these operations. Using the simple application developed in Section 5.4.2, an example is developed through this section.

write This operation stores a new entry into the local space of the host on which the operation is initiated. The produced entry is stored until its lease is expired. It is worth restating that each entry is only stored on the local host and is not replicated over the OppNet. Therefore, it is up to each host to monitor its local space and manage its own entries, and especially ensure that out-of-date entries are removed.

Let us consider the OppNet shown in Figure 5.5, and assume that a user (or a process in JavaSpaces terminology) called *P1* in island 4 has an available seat from *"Paris"* to *"Roma"* on Tuesday 22/09/2015, so it writes the following entry in the space:

```
Seat offer= new Seat("Paris","Roma", new Date(2015,9,22));
```

As mentioned above, the entry *offer* is only stored locally and is not disseminated over the OppNet.

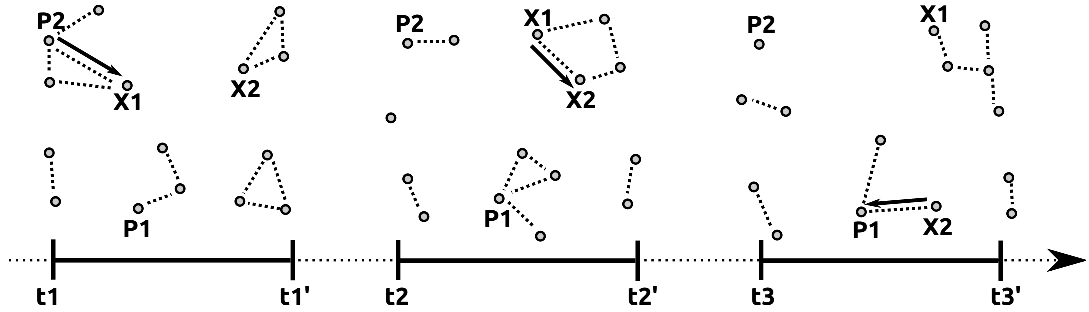


Figure 5.6: Example of a message transmission in an OppNet

read This thread-coupling operation requests the JION service to locate an entry that matches the template provided as a parameter. When a process on a host performs a *read* operation, the local space of the host is queried first in order to find a matching entry. If no matching entry is found, the process is suspended for a certain amount of time specified in the timeout provided as a parameter. Meanwhile, JION disseminates the request over the OppNet, using the timeout value as a deadline for the given request. Each host, when it receives this request, queries its local space to find a matching entry and forwards a copy of this entry (if any) back to the requesting process. The deadline of the response has the same value as the deadline of the request. When JION receives an answer to the *read* operation, it resumes the requesting process, which is then informed about the available answer.

If the timeout value expires and no answer has been received, JION resumes the calling process and returns a *null* as an answer.

Let us consider that a user *P2* in island 1 wants to know when it is possible to go from *"Paris"* to *"Roma"*, so that it provides JION with the corresponding template, which is shown here:

```
Seat request= new Seat("Paris", "Roma", null);
```

In order to transfer this request to *P1*, which has a matching entry, *P2* transmits a copy of the request to *X1*, with which it has a contact during the interval $(t_1,t_1^{'})$ as shown in Figure 5.6. Then, the carrier *X1* moves and becomes connected with *X2* during the interval $(t_2,t_2^{'})$, in which it transmits a copy of the request to *X2*. In the same way, while *X2* is moving, it transmits a copy of the request to the final destination *P1* after an arbitrary disconnectivity interval $(t_2^{'},t_3)$. The process *P1*, when receiving the request, forwards a copy of *offer* back to *P2* using the mobility of hosts between islands. As a result, *P2* knows that a seat for *"Roma"* shall be available on Tuesday 22/9/2015.

Operation *readIfExists* is also supported by JION. According to the JavaSpaces specification, the aim of this operation is to provide an immediate answer. To do so, traditional JavaSpaces implementations query only some central servers and immediately returns either a matching entry or a *null* indicating that no matching entry exists. Taking into consideration the absence of such central servers in an OppNet, the semantic of immediate answer can only be obtained by querying locally. For this reason, the *readIfExists* in JION queries only the local space to find an entry that matches the specified template. The request is not disseminated over the OppNet.

take This thread-coupling operation basically performs the same function as *read*, except that it removes the matching entry from the space. JION first searches the local space. If no match is found, it suspends the process for the timeout value provided as a parameter, while searching for the answer. It starts by querying all the hosts over the OppNet in order to discover which hosts (if any) have a matching entry. The hosts send back proposals to the calling process. Upon receiving the proposals, JION uses its selection policy to select one host, from which the entry should be taken. JION then asks the chosen host to permanently remove the matching entry from its local space and hands it back to the requesting process.

If the timeout expires and no answer has been received, then the operation terminates and the process is resumed to get informed about the issue.

The entire operation may take more time than the *read* operation since it involves up to four message disseminations in the network, while only two messages are required in the *read* operation.

JION also supports a *takeIfExists* operation, which performs exactly like the corresponding *readIfExists*, except that a matching entry is removed from the local space.

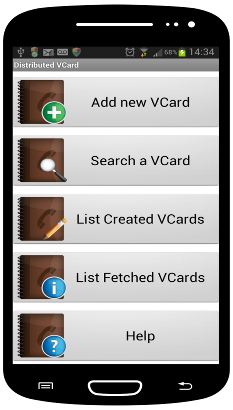
readf and takef These operations are thread-decoupling versions of *read* and *take* operations, respectively. They allow non-blocking access to the space using the technique of Future object. During the whole operation, the calling process continues executing its code without being blocked, as if the operation had been effectively performed, as long as the calling process does not need the real answer. The calling process will only be blocked if it does need the actual answer, and that answer is not yet available.

When invoking *readf* or *takef*, as shown in Figure 5.3.b, a new Future object is created and returned instantly as the result of the invocation. So the calling process is not blocked waiting for the answer. Meanwhile, JION behaves as if it was executing a normal *read* or *take* operation. Upon the completion of the operation, the answer is assigned to the Future object. It is then possible for the calling process to get the result using method *get*.

notify This operation informs a process when entries matching a set template are written in the space. When a process is interested about specific entries, it provides JION with the corresponding template. JION disseminates the template all over the hosts in the OppNet. The hosts register the request in the hope that a matching entry will be written before the lifetime of the request expires. Consequently, when a matching entry is written in a host, the host sends an event object containing information about this entry to the requesting process.

5.7.4 Transactions

According to the JavaSpaces specification, it is possible to group multiple operations into a transaction, that acts as a single atomic operation. This is done using the optional concept of transaction. Using JavaSpaces terminology, once a client joins a transaction, it becomes a *participant*. Either all participants' operations within the transaction are performed, or none is.



(a) D-vCard Android application

```
BEGIN:VCARD
VERSION:4.0
FN:Abdulkader BENCHI
N:BENCHI;Abdulkader;;;
ORG:IRISA
NOTE:vCard example
END:VCARD
```

(b) vCard example

Figure 5.7: Distributed application based on JION

In a fully-connected network, a transaction is controlled by a specific manager (server), which should always be reachable by all the participants. If a participant is momentarily disconnected, the whole transaction is aborted. Considering that hosts in OppNets cannot rely on a reliable server, can become disconnected and their messages can be lost, it is not possible to ensure transactions as defined by JavaSpaces. However, transactions could be supported alternatively by leveraging a disruption-tolerant consensus implementation. By doing so, participants in a given transaction could agree between themselves (i.e., without leveraging a manager) whether to perform all operations under this transaction or to abort the transaction.

The current version of JION does not support the concept of transaction. Using JavaSpaces terminology, each operation in JION is considered as a singleton operation[4].

5.8 Application Case Study: Distributed vCards Directory System (D-vCard)

For testing and evaluation purposes, a distributed vCards directory system (D-vCard) has been developed based on JION for Android platforms, as shown in Figure 5.7a. D-vCard is a distributed address book that allows users of Android smartphones to access and opportunistically share contact data between their smartphones. D-vCard complies with the vCard electronic business card specification, as defined in RFCs 6350 [93]. A vCard is a format specification for representing the kind of information required for every contact data in an address book or email application: name, address information, phone numbers, etc. An example of a vCard is shown in Figure 5.7b. It basically consists of field-value pairs separated by ":". The specification defines all possible fields and the format for their values.

In the application D-vCard, creating a new vCard is simply implemented using *write* operation of JION: a vCard entry is created and stored locally on the space, which acts as

[4]A singleton operation in JavaSpaces terminology denotes an atomic operation

a local directory. Similarly, *searching/deleting* a vCard in/from the space is performed by creating a template having the required information as fields. This template can then be passed as a parameter to the *readf/takef* operation, respectively. Once a user gets a vCard, he/she will automatically receive notifications about any new modification on the given vCard thanks to the *notify* operation.

5.9 Evaluation

The originality of this work lies in the fact that JION has been fully implemented in Java, and can thus be tested in real-life experiments on Linux and Android platforms. JION is distributed under the terms of the GNU General Public License[5]. It has seen extensive testing to examine how it performs in OppNets. The conducted tests strived to evaluate how easy it is for an application developer to implement a distributed application using JION. Furthermore, they have evaluated the efficiency of JION in real conditions.

5.9.1 Developing Distributed Applications with JION

JION implements Sun Microsystems' JavaSpaces Technology specification, provided as a part of the Java Jini Technology [46]. Since it implements a well-known middleware specification, a developer can simply focus on writing a standard JavaSpaces application, and JION will take care of its execution in an OppNet. Indeed, any pre-existing JavaSpaces application can be deployed using JION.

As a proof-of-concept, an already-existing JavaSpaces application has been deployed successfully over JION. The whole source code is provided in Code 5.5. The import statements are not shown and the class *Seat* is defined in Code 5.2. Obtaining access to a space is the start point for a JavaSpaces application to interact with the space (line 8). Each JavaSpaces implementation provides a different means of accessing spaces. For JION, this is done using the *JavaSpaceSingleton* class (line 4).

Assuming that JION's package is installed in the home directory, this example can be compiled with the following command:

```
% javac -cp $HOME/JION.jar HelloWorld.java
```

Once the example has been successfully compiled, it can be run on a sender-side terminal in the following manner:

```
% java -cp $HOME/JION.jar HelloWorld write
```

Similarly on a receiver-side terminal for the *read* operation:

```
% java -cp $HOME/JION.jar HelloWorld read
```

Likewise, the *take* operation could be run on a receiver-side terminal in the following manner:

```
% java -cp $HOME/JION.jar HelloWorld take
```

[5] http://www-casa.irisa.fr/jion/

Code 5.5 Deploying an already-existing JavaSpaces application over JION

```
public class HelloWorld{
    public static JavaSpace getSpace(){
        // return an instance of a space
        return JavaSpaceSingleton.getInstance();
    }
    public static void main(String[] args) throws Exception{
        // obtain access to a space
        JavaSpace space = getSpace();
        // write operation
        if (args[0].equals("write")){
            Seat offer = new Seat("Paris", "Madrid", new Date(2015,9,22));
            space.write(offer, null, Lease.FOREVER);
        }
        // read operation
        else if (args[0].equals("read")){
            Seat template = new Seat();
            Seat result = (Seat) space.read(request, null, Lease.FOREVER);
            System.out.println(result);
        }
        // take operation
        else if (args[0].equals("take")){
            Seat template = new Seat();
            Seat result = (Seat)space.take(request, null, Lease.FOREVER);
            System.out.println(result);
        }
    }
}
```

5.9.2 Experimental Evaluation

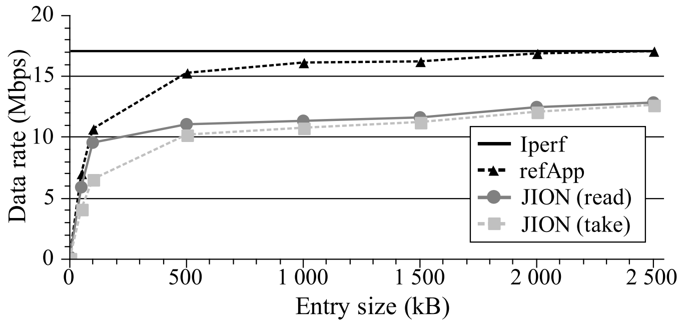
The measurement campaigns presented in Section 4.8.2 and 4.8.3 are also conducted here in order to evaluate the performance of JION. However, the second measurement campaign involved several user-carried smartphones running D-vCard. During these campaigns, the performance of JION has been compared against two other tuple spaces implementations: TB (Tuple board) [94] and LIME (Linda in a Mobile Environment) [95]. Using the collected traces, several performance metrics have been computed. The campaigns along with their results are described in details in the next sections.

During this evaluation, it would have been interesting to compare the performance of JION against more other JavaSpaces providers. Unfortunately, beside these implementations no other JavaSpaces implementation is openly-distributed in the context of mobile ad hoc environments.

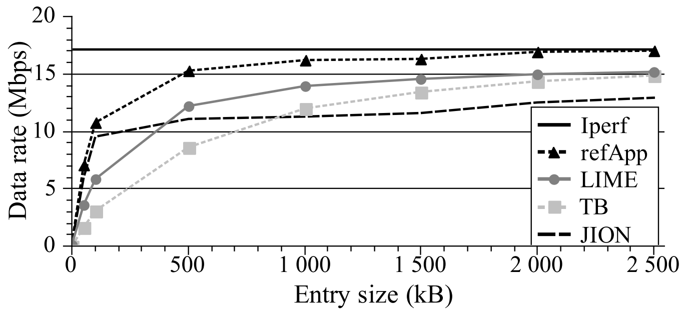
5.9.2.1 Efficiency of JION over a Single Connected Island

The main objective of the first campaign was to evaluate the performance of the multilayer architecture introduced in JION, over the underlying wireless transmission medium.

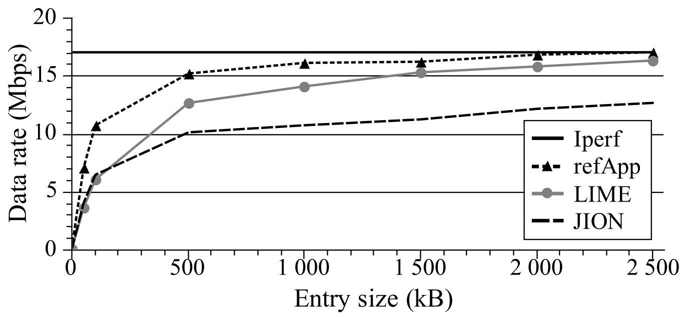
This campaign started by setting up two netbooks A and B using the same configuration presented in Section 4.8.2. They were running JION, TB and LIME over a Linux operating system. The test actually focused on the response time, which is here defined as the time interval between the time netbook A sends a request to read/take entries of different sizes to netbook B and the time when it actually receives them. In order to get



(a) Performance of JION against baseline measurement tools



(b) Performance of read operation using different implementations



(c) Performance of take operation using different implementations

Figure 5.8: Transmission throughputs observed between two hosts

baseline values, the network performance measurement tool *Iperf* along with the *refApp* presented in Section 4.8.2 are used.

A total of 750 series of tests were conducted in this scenario (250 series with each tuple spaces implementation). The results of these tests are presented in Figure 5.8a, 5.8b and 5.8c.

As shown in Figure 5.8a, JION shows a similar performance for both read and take operations. However, JION shows about 20% overhead over *refApp*, which is an indication of the cost of discovering neighbors and gossiping with them before exchanging actual applicative messages.

The results shown in Figure 5.8b and 5.8c show that, surprisingly, take operation of LIME performs slightly better than its read operation. They also show that, as could be expected, the performance of a tuple space implementation depends on the size of tuples.

On the one hand, JION performs better than TB and LIME for small-sized tuples. This is due to the fact that both TB and LIME rely on TCP transmissions, so that they must establish a connection every time they send tuples. Establishing these connections in order to exchange small-sized tuples represents non-negligible performance overhead. During the experience, netbook A created a small tuple with the size of 1kB, and netbook B tried then to read it using each tuple spaces implementation. The average response time was about 41.1 ms with JION, 167.67 ms with LIME and 262.24 ms with TB.

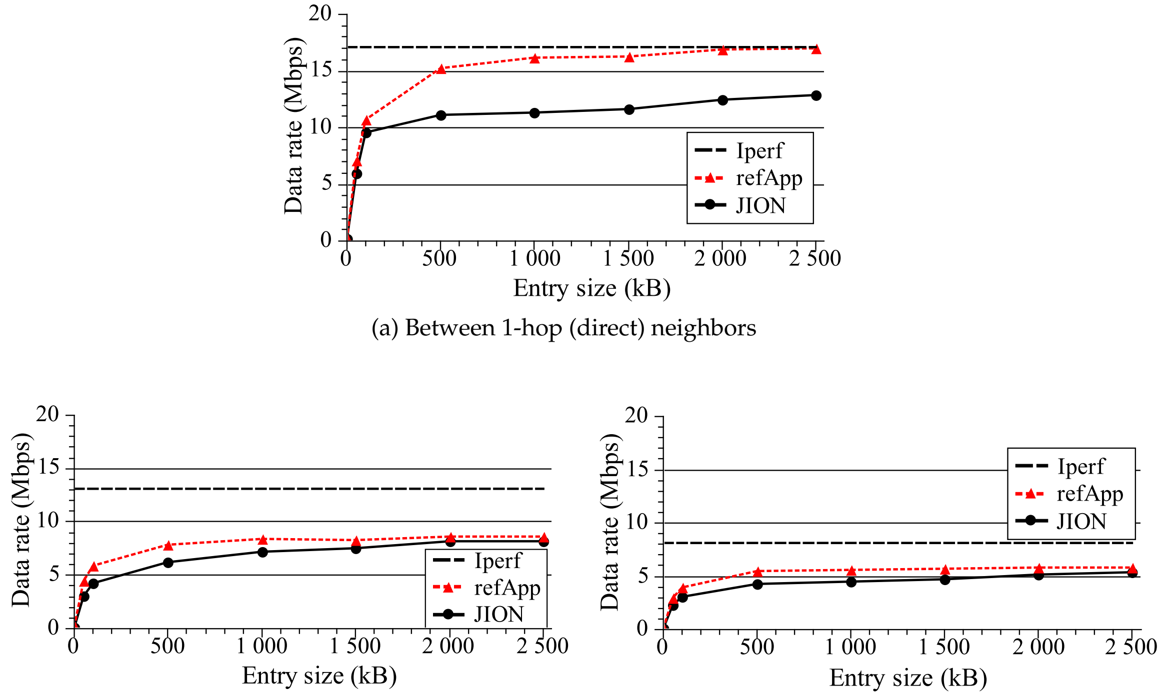
From the other hand, JION shows about 15% overhead over LIME and about 10% overhead over TB for big-size tuples. Considering that both TB and LIME are specifically designed to work in such fully-connected scenarios, whereas JION is designed to work in OppNets by implementing a sophisticated opportunistic protocol in order to orchestrate communications between neighbor hosts, results can still be considered quite reasonable.

It is worth noting that the current implementation of TB does not support the *take* operation, that is why there is no corresponding curve for TB in Figure 5.8c.

During the previous test, the delay to produce a tuple is also measured. To do that, the time elapsed between two moments is calculated: the moment when a tuple is sent at the application layer, and the moment in which the tuple is actually sent at the physical layer. The obtained results have shown an average latency of 10 ms with *refApp*, 22 ms with JION and 25.8 ms with TB. As for LIME, the average latency varies between 40 ms (stable situation) and 1228.2 ms (some unannounced disconnections appear in the system). This is discussed further in Section 5.11.

Another test was conducted to investigate the behavior of these implementations when entries can propagate among connected devices, that is, within a single connectivity island. The test was carried out with four netbooks (A, B, C and D) distributed following the same schema presented in Section 4.8.3. These netbooks were configured using the OLSR routing protocols so that to ensure that every netbook can reach each other. This test relied on *write/read* operations: a total amount of 225 entries were written on B, C, D, and host A was configured to read these entries using each JavaSpaces implementation, separately. During this test, the average time required for these entries to reach host A is measured. As a baseline, the *Iperf* and *refApp* are also used to measure the throughput achievable at message level across multi-hop wireless links.

Surprisingly, both of LIME and TB were not able to perform *read* operation. In fact,



(a) Between 1-hop (direct) neighbors

(b) Between 2-hop neighbors

(c) Between 3-hop neighbors

Figure 5.9: Transmission throughputs observed in a single connected island (with or without multi-hop forwarding)

they are not designed to get advantage from multi-hop communications. In other words, these implementations can only work in a group of *directly-connected* hosts. More details about these implementations are presented in Section 5.11.

The results of JION are shown in Figure 5.9a, 5.9b and 5.9c for transmissions over 1, 2 and 3 hops, respectively. As could be expected, these results are nearly the same as those obtained using *receive* operation in JOMS, as presented in Section 4.8.3. Indeed, both of *read* operation in JION and *receive* operation in JOMS function in a similar manner: they leverage *subscribe* method provided by DoDWAN to receive the required information, which is then processed locally.

It can be noticed that the delay before netbook A gets an entry changes with the size of the entry (as could be expected), but also with the number of transmission hops. This is because when netbook B serves as a relay between its neighbors A and C, the radio channel around B is twice as busy as when B interacts only with host A. The same observation applies for netbook C when it must serve as a relay between netbooks B and D.

It can be also observed that JION shows 7% to 26% overhead over *refApp*, which is an indication of the cost of discovering neighbors and gossiping with them before exchanging actual applicative messages. This overhead gets lower when multi-hop forwarding is required, for in that case *refApp* must receive messages before forwarding them, just like JION.

It must also be considered that JION using the underlaying communication layer

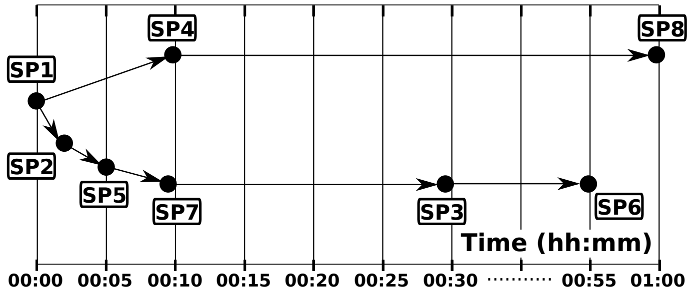


Figure 5.10: Timeline of the dissemination of a message

strives to ensure a high delivery ratio, so entries are retransmitted if they are not received in the first place. Indeed, during the test all tuples got received by netbook A, so that the delivery ratio was 100% in that case.

Of course, JION's most salient feature is that it can ensure the delivery of messages in an opportunistic network, where other tuple spaces implementations designed for fully-connected environments (e.g., LIME and TB, just to name a few), as well as simple applications like *refApp*, are totally useless.

5.9.2.2 Efficiency of JION in a Real OppNet

The previous tests demonstrated that JION performs satisfactorily in a traditional connected environment and provides a reasonable response time. However, a disruption-tolerant system such as JION is not really necessary in a connected environment. Indeed, it becomes really beneficial when the network shows connectivity disruptions, as shown in Figure 5.5.

Consequently, after measuring how JION can perform in a single connected island, the D-vCard application presented in Section 5.8 was used to observe how JION can perform in a real OppNet. Eight volunteers from IRISA laboratory were equipped with HTC smartphones running D-vCard. During this campaign, which lasted for 8 hours, the volunteers were asked to carry their smartphones whenever possible –and use D-vCard services of course– while roaming the laboratory building or its surroundings.

Figure 5.10 illustrates how one particular "notify request" message disseminated during one of the trials. This message was first published by smartphone *SP1*. After only a few minutes *SP1* established radio contact with *SP2*, which thus got a copy of the message and became a new carrier for this message. *SP1* later managed to forward the message to *SP4*, while *SP2* forwarded it to *SP5*. The message thus kept disseminating, until it reached the last smartphone *SP8*, about one hour after it was initially published. This example confirms the delay/disruption-tolerant nature of transmissions.

An important factor in OppNets is how often the mobile hosts meet each other. During this campaign, the cumulative distribution function (CDF) of the average number of neighbors perceived by all smartphones is shown in Figure 5.11. It can be noticed that the average number of neighbors for each smartphone was about two neighbors during about 65% of the campaign duration and no neighbor at all during 25% of the campaign duration. A finer detail of smartphone density, as observed during the campaign, is pro-

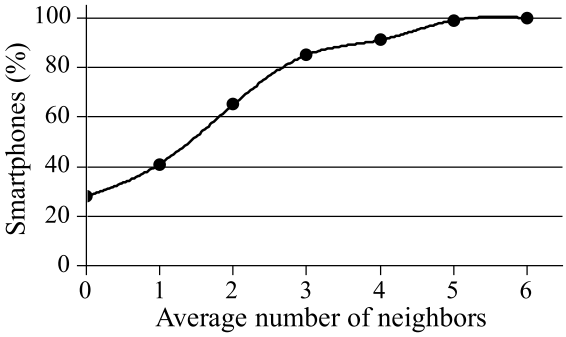


Figure 5.11: Cumulative distribution function of the average number of neighbors perceived by all smartphones

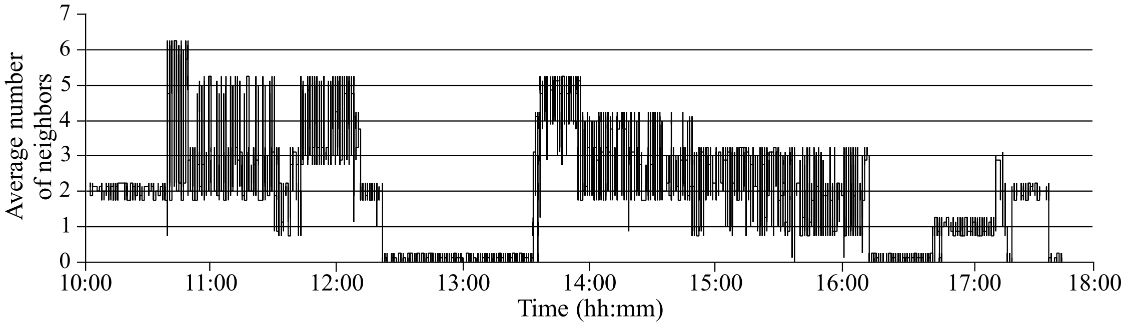


Figure 5.12: Timeline of the average number of neighbors during the campaign

vided in Figure 5.12. It is worth noting the frequent disconnections between smartphones during the majority of the campaign duration, which makes a server-based system with synchronous operations totally impractical, and demonstrate the need for opportunistic transmissions in order to maintain communication in such conditions.

When analyzing the traces collected by JION, special attention was paid to the delivery time of successful operations. The delivery time is calculated as the difference between the time when an operation is started and the time when an answer is received. The delivery times are then represented in cumulative way as shown in Figure 5.13. The cumulative delivery times were calculated in terms of time slices where a measure of 5 minutes means that the average delivery time observed was between 0 and 5 minutes;

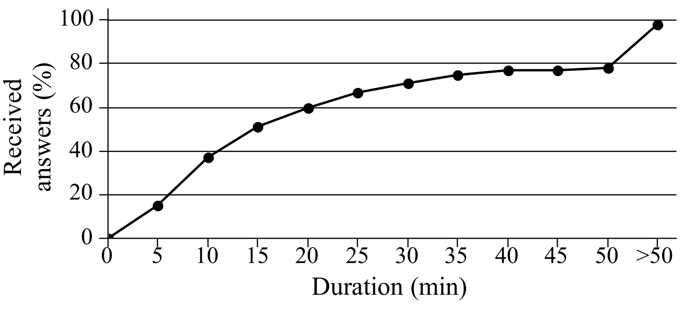


Figure 5.13: Cumulative distribution function of the delivery times of received answers

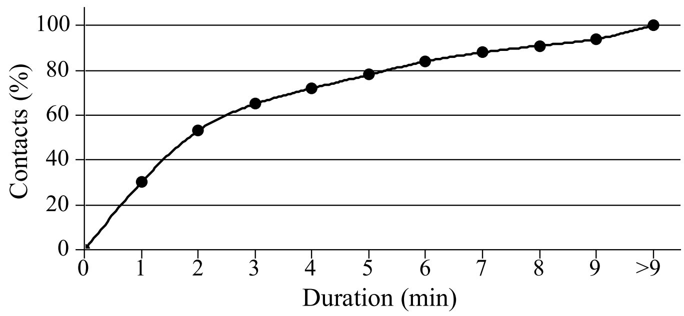


Figure 5.14: Distribution function of contact duration between neighbors

a measure of 10 minutes means it was between 0 and 10 minutes, etc. In general, the results show that nearly 80% of the entries got delivered to their destination(s) in less than 50 minutes. Yet, about 3% of the entries could not be delivered. This is the consequence of the unpredictable –yet perfectly legitimate– behavior of the users, which sometimes moved away from the laboratory or switched their devices off in order to preserve their battery budget. By doing so they prevented any further radio contact between their device and those of other users, and this of course led to message loss.

During this campaign, a total number of 3806 radio contacts were established between neighbor smartphones, with an average contact duration of 213 seconds. The contact durations are calculated in terms of time slices and presented in Figure 5.14. The majority of radio contacts lasted less than a minute, which confirms the continuous topological changes of the network during the campaign. Hence, systems that want to rely on these contacts to transport data between smartphones, have to resort to DTN-style communication.

Another metric that was also calculated during this campaign is the effect of multi-hop forwarding. Figure 5.15 shows the percentage of direct/multi-hop entries. Direct-entries (shown as 1-hop in the figure) are entries that have been directly received from their original source. Multi-hop entries are entries that have been relayed by several intermediate smartphones before reaching their final destination. For example, the smartphone presented as SP4 in Figure 5.15 received its entries in the following manner: 51% were received directly from the sender (1-hop), 40% through 2 hops, 6% through 3 hops, 2% through 4 hops and finally 1% through 5 hops. It can be observed that around 50% of the entries were received in a multi-hop manner, showing that JION using the underlying communication protocol provides a multi-hop dissemination service, that performs satisfactorily in a real OppNet.

5.10 Discussion

The results presented in Section 5.9 confirm that JION is quite efficient in providing JavaSpaces services in OppNets.

An obvious advantage of implementing a provider that conforms with the JavaSpaces standard specification is that developers do not need to learn a new programming language, or get familiar with an exotic programming model or API. Indeed, any pre-

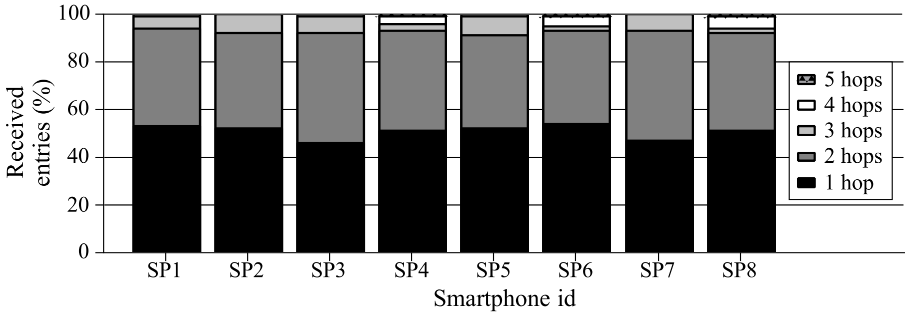


Figure 5.15: Amount of entries received in a direct/multi-hop manner per smartphone

existing JavaSpaces application can be deployed over OppNets easily using JION.

To assess the advantages of using Future object, it is worth noting that performing blocking operations, as provided in standard JavaSpaces implementations, is not preferable in OppNets. Indeed, it is impractical for an application to be blocked waiting for an answer for an unknown amount of time. In such case, it will be always considered as an *unresponsive* application. Each system deals with such unresponsive application differently. A standard way to overcome this limitation of blocking operations is often to design multi-threaded applications. However, the use of Future object, from a readability point of view, is quite important since it permits to make the code more *readable,* more structured and therefore more intuitive. For the sake of illustration, Code 5.6 presents a *non-blocking* JavaSpaces application written using the technique of *Threads*. Note that the class *Seat* is defined in Code 5.2. The previous code can be made more readable by changing the lines, from 11 to 51 by Code 5.7, which involve using operations based on Future object. Comparing the two versions of the application, one can argue that non-blocking JavaSpaces application can be easily achieved using Future object while keeping its source code human-readable and maintainable as well.

5.11 Related Work

In the last few years, several projects revisited Linda [88], especially in the context of mobile ad hoc environments.

Both Ara [96] and LIME [95] are coordination middleware systems implementing tuple spaces stored on hosts acting as servers. These middleware systems target mobile ad hoc networks. They provide JavaSpaces services to mobile hosts that are in communication range of the servers. Since they rely on a server-based model, they are hardly usable in real OppNets.

LIME is openly-distributed for the developers that work in the context of MANETs [6]. The current version of LIME assumes that all hosts in an ad hoc network are directly connected to one another. In other words, all hosts form a kind of group, in which there is only one hop between any pair of host. This group of hosts is managed by a leader, which plays a major role in the overall LIME architecture design. Every time a change occurs in

[6] `http://lime.sourceforge.net/`

Code 5.6 Asynchronous programming of a JavaSpaces application using Thread technique

```
public class NonBlocking {
    static Seat firstRequest= null;
    static Seat secondRequest= null;
    static long ttl=Lease.FOREVER;

    public static void main(String [] args){
        final JavaSpace space= JavaSpaceSingleton.getInstance();
        final Message firstTemplate= new Seat("Paris", "Roma", null);
        final Message secondTemplate= new Seat("Paris", null, new Date(2015,9,22));

        // first non-blocking read operation
        Thread firstOperation= new Thread(){
            public void run(){
                try{
                    synchronized (firstTemplate) {
                        firstRequest= (Seat) space.read(firstTemplate, null, ttl);
                    }
                } catch (Exception e){
                    e.printStackTrace();
                }
            }
        };

        // second non-blocking read operation
        Thread secondOperation= new Thread(){
            public void run(){
                try{
                    synchronized (secondTemplate){
                        secondRequest= (Seat) space.read(secondTemplate, null, ttl);
                    }
                } catch (Exception e){
                    e.printStackTrace();
                }
            }
        };

        // start operations
        firstOperation.start();
        secondOperation.start();

        {
        // developers can here perform other tasks while waiting the answers
        }

        // print answers
        synchronized (firstTemplate){
            System.out.println(firstRequest);
        }
        synchronized (secondTemplate){
            System.out.println(secondRequest);
        }
    }
}
```

Code 5.7 Asynchronous programming of a JavaSpaces application using Future objects

```
try{
    // start Future operations
    Future<Seat> firstRequest= space.readf(firstTemplate, null, ttl);
    Future<Seat> secondRequest= space.readf(secondTemplate, null, ttl);

    {
    // developers can here perform other tasks while waiting the answers
    }

    // print answers
    if (firstRequest.isDone())
        System.out.println(firstRequest.get());
    if (secondRequest.isDone())
        System.out.println(secondRequest.get());

} catch (Exception e){
    e.printStackTrace();
}
```

a group (i.e., a host joins/leaves the group), the other hosts of the group perform a leader election in order to elect a new leader for this group. During the election procedure which takes about one second, all running operations stay blocked. Arbitrary connections and disconnections are not supported in the current implementation of LIME. Changes in connectivity must be *explicitly* notified by users using the API. When an unannounced disconnection appears in a group during an operation, the whole system gets hanged for a couple of seconds until this unannounced disappearance is detected. Similarly, when a host connects during an operation, it will be ignored for this operation. The current implementation of LIME is not a crash-tolerant system: all tuples are only stored in a volatile storage. When a host is switched off, all the stored tuples are immediately deleted.

Limone [97] is a lightweight alternative to LIME imposing far less overhead. It is based on the premise that a single round-trip message exchange is always possible, making it impractical over OppNets.

CAST [98] is a server-less coordination middleware for MANETs. Since it does not rely on any centralized service, this middleware suits well the dynamics of wireless open networks. CAST makes it possible to process operations even when no end-to-end route exists between the hosts involved, by implementing a source routing algorithm. This routing strategy relies on the assumption that the motion profile of each host is known. This is clearly a serious constraint, which limits the usability of CAST over the kind of OppNets JION targets, where the motions of hosts are neither planned nor predictable.

Tuple board (TB) [94] is another server-less coordination middleware for developing collaborative applications running in ad hoc networks of mobile computing devices. Like JION, this middleware has been fully implemented and distributed[7]. The key difference between the tuple board and tuple space is that tuples do not persist on the tuple board if the posting process goes away. In fact, the proposal is limited to a group of directly connected devices: when a device leaves the network or turns off, all the tuples posted by this device are withdrawn. By delving into the finer details, the current version of TB

[7] `http://www.cs.rit.edu/~ark/tb.shtml`

Table 5.1: Comparison between the reviewed JavaSpaces Providers

JavaSpaces Provider	Multihop forwarding	Disruption tolerance	Peer-to-peer	Unplanned mobility	Take operation	Asynchronous operations	Openly distributed
ARA [96]	X	X	X	✓	✓	X	X
LIME [95]	X	X	X	✓	✓	✓	✓
LIMONE [97]	X	✓	✓	✓	✓	✓	X
CAST [98]	✓	✓	✓	X	✓	X	X
TB [94]	X	X	✓	✓	X	X	✓
JION	✓	✓	✓	✓	✓	✓	✓

is implemented over Many-to-Many-Invocation (M2MI) [99], a Java distributed object middleware system for ad hoc collaborative applications. M2MI provides a broadcast remote method invocation capability between nearby networked devices. Formally speaking, M2MI can only provide wireless proximal services between devices which are within transmission range of each other. Such a communication layer prevents TB from being used in OppNets, where the frequency of link disruptions makes the disconnection tolerance a vital requirement for any middleware that is meant to support these networks. Furthermore, the current implementation of TB is not crash-tolerant where all tuples are only stored in a volatile storage.

It can be noticed that all these middleware systems, except LIME and Limone, are not well-suited for OppNets, as they do not support asynchronous operations. LIME and Limone, as traditional JavaSpaces implementations, use a basic approach where a timeout can be specified for their blocking operations to avoid locking the system indefinitely. Using the timeout technique, JavaSpaces implementations stay blocked waiting for the timeout period to be elapsed, making them impractical in OppNets.

Furthermore, all middleware systems mentioned in this section, except TB, define their own communication protocol. JION is different as it presents a two-layer architecture: the upper layer is concerned with tuple space services, while the lower layer supports opportunistic communication. As mentioned in Chapter 3, DoDWAN has been chosen among a few opportunistic communication protocols that are openly distributed to support communications on OppNets. Yet, JION could theoretically be implemented above any other opportunistic communication system.

To stress the main difference of the aforementioned JavaSpaces providers, a comparison table is presented in Table 4.1. This table presents whether multihop forwarding and disruption tolerance are supported in a given JavaSpaces provider. Furthermore, it presents the way by which a JavaSpaces provider deals with hosts: whether they are all equal peers or they follow the hierarchical roles of master and slave. The table presents whether the mobility pattern of mobile devices is planned or not. The table also presents whether *take* operation and asynchronous operations are supported in a JavaSpaces provider. Finally, it presents whether a JavaSpaces provider has been openly distributed and is thus accessible to developers, or it is not distributed and cannot be used in real conditions.

5.12 Summary

The tuple space programming model is interesting for both communication and coordination in distributed applications, and the JavaSpaces technology provides a functional implementation of this programming model for Java applications. Yet most current JavaSpaces implementations are server-based systems. Since no host in an OppNet can be considered as stable and accessible enough to play the role of a server for all other hosts, a server-less implementation of the JavaSpaces specification is required for applications targeting OppNets.

This chapter presented JION (JavaSpaces Implementation for Opportunistic Networks), a JavaSpaces implementation with support for Future object, designed and implemented specifically for OppNets. Using JION, distributed applications based on the tuple spaces programming model (as defined in the JavaSpaces specification) can be deployed and executed in OppNets. JION is resilient to the challenges of OppNets and provides an effective base which eases the development of applications for OppNets. It has been tested on Linux and Android platforms in real conditions and is now distributed under the terms of the GNU General Public License.

An important issue has emerged while developing middleware systems in the context of OppNets. The lack of a openly-distributed disruption-tolerant consensus implementation in the context of OppNets makes it impossible to support several concepts that must be managed collaboratively. Indeed, mobile devices form an OppNet without the help of any system, that allows them to decide collaboratively. Such an issue inherently fosters mobile devices to behave individually. Inducing a consensus implementation in OppNets paves the way for supporting coordination between mobile devices in these networks. This represents the main research topic of the next chapter.

6

Solving Consensus in Opportunistic Networks

Contents

ENSURING the coordination of multiple parts of a distributed application in OppNets, in which pairwise unpredicted transient contacts between mobile devices are the only opportunities for these devices to exchange information or services, is a challenge. The absence of a coordination system that helps mobile devices to decide collaboratively in OppNets, fosters these mobile devices to behave individually.

This chapter starts by briefly reviewing the consensus problem in general. It then goes on to examine how a consensus can be solved in an OppNet. As a proof-of-concept, a fully-implemented system is designed along this chapter. Its architecture combines an implementation of a variant of the One-Third Rule (OTR) algorithm with DoDWAN protocol. Experimental results and simulations are also presented, that validate the system and demonstrate that consensus can be solved effectively in an OppNet.

6.1 Introduction

The middleware systems proposed in the former chapters constitute a first step toward application development targeting OppNets. Yet, in complex distributed applications, multiple processes running on different mobile nodes must be able to agree on a common decision.

Consensus is an essential building block for distributed applications, as it facilitates consistent distributed behavior. Consensus allows processes to reach a common decision, which depends on their initial inputs despite failures, and can be used to solve many related problems such as group membership, leader election, transaction commitment, etc. [100].

Consensus would allow to redesign the middleware systems presented in the former sections. Indeed, it could allow them to support their operations differently. For example, JION can leverage a disruption-tolerant consensus system to support the tuple space programming model differently by which the *write* operations are disseminated network-wide and a *take* operation for a given entry calls for a consensus to agree to delete the entry network-wide. Consensus would also facilitate the development of self-adaptive JMS and JavaSpaces applications, that can agree to perform some adaptations based on some criteria (e.g., migrating a queue manager between mobile devices in JOMS when resources are not sufficient).

Consensus problems have been studied extensively in the literature, with various system models. Each system model makes specific assumptions about the underlying communication model (synchronous or asynchronous, based on either reliable or faulty links) and about the processes themselves (reliable, susceptible to crash or to exhibit Byzantine failures). Most papers addressing consensus adopt system models that suit traditional wired networks, such as the Internet. To date only a few papers have considered the consensus problem in wireless ad hoc networks and no paper has considered the consensus problem in OppNets yet.

As observed in [101] the models for wired networks are often strongly biased towards node failures to the detriment of link failures. Yet, mobile ad hoc networks –including OppNets– require a model that admits both transient process and link faults. The Heard-Of (HO) model [102] meets these requirements. This model does not distinguish faulty processes from faulty links, as it focuses on transmission faults (effects) without accounting for the faulty components (causes) [102].

As discussed in Section 2.2.2, in an OppNet a wireless link between two neighbor nodes is inherently transient, so when this link disappears (for example because both nodes have moved away from each other) this should not be considered as a "fault": this is the expected consequence of both mobility and limited transmission range in an OppNet. Similarly, mobile nodes in an OppNet often run on batteries, so a common strategy to preserve their power budget is to turn them off –or put them in suspend mode, or disable their wireless transceiver– frequently. When a node suddenly "disappears" from the network, this should not always be considered as a "fault", because this is a perfectly legitimate behavior for this kind of node in this kind of environment.

As mentioned above, delivering messages to remote nodes in an OppNet is not guaranteed, for it depends on the wanderings of carriers whose mobility patterns are neither planned nor controlled. Messages can therefore get lost before reaching their destination(s). Two of the consensus algorithms defined in [102] for the HO model can easily tolerate message loss: the Paxos/LastVoting (P/LV) algorithm, and the One-Third Rule (OTR) algorithm. An implementation of the P/LV algorithm for mobile ad hoc networks has been proposed in [101]. This implementation could not run in an OppNet, though, as it is a server-based implementation that requires temporaneous end-to-end connectivity between the coordinator (i.e., the server) and all the other processes.

This chapter presents ADAM (Agreement in Disconnected Adhoc Mobile networks), a middleware system that implements a variant of the OTR algorithm, and that can run effectively in an OppNet. ADAM has been fully implemented in Java, and it has been tested in real conditions using a small flotilla of smartphones as mobile nodes. ADAM has also been tested using simulation so as evaluate its performance in a large-scale network involving a larger population.

In ADAM, the OTR algorithm is implemented on top of DoDWAN. This combination of the OTR algorithm and the epidemic routing supported by DoDWAN is consistent, because epidemic routing is an effective way to disseminate messages in an OppNet, where mobility patterns are neither planned nor controlled, and where mobile nodes (with the messages they carry) can disappear for a while from the network. Besides, the OTR algorithm requires *n-n* communication: in every step each process (or mobile node) must send a message to all other processes (or nodes). With epidemic routing, sending a message to all nodes is not significantly different from sending a message to a single node, so combining the OTR algorithm with epidemic routing makes sense when targeting OppNets.

6.2 Background

The problem of consensus can be informally stated as follows: each node proposes a value and has to decide a value such that all nodes have the same decided value, which is one of the proposed values. The consensus problem has been studied in various system models, characterized by their synchrony (synchronous/asynchronous systems) and failure models (process/link failures). The design of consensus algorithms is closely related to the underlying system model: a consensus algorithm specifically designed for synchronous systems is not going to work in asynchronous systems.

The consensus problem over a set $\Pi = \{p1, p2, ...p_n\}$ of processes (which are called *participants* in the system model introduced in this chapter and are assumed to run on distinct nodes) is defined by the following three correctness properties:

- Validity: Any decision is the initial value of some participant.
- Agreement: No two participants decide differently.
- Termination: All correct participants[1] eventually decide upon some value.

The termination property defines the *liveness* associated with the consensus system, while the agreement property and validity property define the *safety* associated with the consensus system.

In the consensus literature, researchers encounter a fundamental result in the theory of asynchronous distributed system: the net effect of the uncertainty in terms of timeliness combined with the uncertainty in terms of failure actually create an uncertainty on the state of the system (as perceived by each node). The latter makes it impossible to

[1] A *correct participant* is a participant that does not crash permanently during a consensus session.

guarantee that a consensus can be solved. This well-known impossibility result is usually refereed to as the FLP impossibility result [103], after its authors' names.

The FLP impossibility result, which albeit being a negative result, ended up being a driving force of researches in the area. The word *guarantee* in the statement of the impossibility result does not mean that nodes can *never* reach consensus in asynchronous system if one is faulty. It allows that a consensus can be reached with some probability greater than zero, confirming what we all know in practice. Indeed, despite the fact that our systems are often effectively asynchronous and their communication systems are not always reliable, these systems have been reaching consensus regularly for many years. It is on this basis that a numerous consensus algorithms have been published in a variety of models to solve consensus in an asynchronous system by *circumventing* the FLP impossibility result. That is, they try to slightly modify either the system model or the problem definition considered in the FLP paper in order to change the conditions in which the FLP result was proven. As a result, the FLP impossibility simply no longer applies and the consensus problem becomes solvable.

6.3 System Model

In an OppNet, no guarantee can be provided about message delivery ratios, transmission delays, node availability, etc. A few assumptions can however be made to the system model presented in Section 2.2.2 so as to limit some of its uncertainties.

In the remainder of this chapter the term *session* is used to call the process that consists in starting a consensus agreement procedure, and pursuing this procedure until a decision is made. Several sessions may of course progress simultaneously in the network, and some nodes may participate in several consensus sessions simultaneously.

It is assumed that a consensus session $\mathcal{S}$ involves a subset $\mathcal{V}_\mathcal{S}$ of the total set of nodes $\mathcal{V}$ such that $\mathcal{V}_\mathcal{S} \subseteq \mathcal{V}$ (*i.e.*, all nodes in the network are not necessarily involved in $\mathcal{S}$). Nodes in $\mathcal{V}_\mathcal{S}$ are called *participants* for session $\mathcal{S}$. Each of these participants is expected to provide an initial value for $\mathcal{S}$, and to run the consensus algorithm until a decision is made. It is assumed that any node $u \in \mathcal{V}_\mathcal{S}$ knows that it belongs to $\mathcal{V}_\mathcal{S}$, and knows $|\mathcal{V}_\mathcal{S}|$ (*i.e.*, how many nodes participate in $\mathcal{S}$). The definition and creation of the subset $\mathcal{V}_\mathcal{S}$ is application-dependent, and is therefore out of the scope of this system. However, the system can be augmented with some mechanisms so as to ease creating $\mathcal{V}_\mathcal{S}$, as discussed in Section 6.9.

Nodes in $\mathcal{V}_\mathcal{S}$ are all expected to serve as mobile carriers for messages pertaining to $\mathcal{S}$. Additionally, any node $u \in \mathcal{V} \setminus \mathcal{V}_\mathcal{S}$ can also serve as a *mule* for messages pertaining to $\mathcal{S}$. Mules that can carry messages pertaining to $\mathcal{S}$ they are not directly interested in thus belong to a subset $\mathcal{V}_{\mathcal{M}(\mathcal{S})}$ (mules for session $\mathcal{S}$).

A consensus session $\mathcal{S}$ spans over a time period $\mathcal{T}_\mathcal{S}$. During some periods in $\mathcal{T}_\mathcal{S}$, some nodes in $\mathcal{V}_\mathcal{S}$ may disappear temporarily from the network. It is assumed that these nodes will eventually reappear during $\mathcal{T}_\mathcal{S}$, and resume their activity regarding $\mathcal{S}$. Some nodes in $\mathcal{V}_\mathcal{S}$ may also crash and disappear definitively from the network. It is assumed that the number of crash failures for nodes in $\mathcal{V}_\mathcal{S}$ is unknown but can be bounded, and that the bound is low with respect to $\mathcal{V}_\mathcal{S}$.

The receive omissions admitted in our system model presented in Section 2.2.2 are here considered as benign faults. The receive omissions ratio is assumed to be low and bounded, though. The system model does not consider Byzantine faults: an active node is assumed to behave properly, and to recover appropriately after being switched off and on again. Furthermore, messages transferred wirelessly between neighbor nodes are assumed to be unaltered. More specifically, whenever a message is received from a neighbor node, the integrity of the received copy is checked, and this copy is simply discarded if it has been altered during the transmission. The communication model ensures that the receiver will find other opportunities to obtain the message anyway (possibly again from the same neighbor).

Finally, for a given node $u \in \mathcal{V}_\mathcal{S}$ it is assumed that there are some periods in $\mathcal{T}_\mathcal{S}$, called "good periods for u" during which all the messages sent by u eventually reach all other nodes in $\mathcal{V}_\mathcal{S}$ (except nodes that crashed during $\mathcal{T}_\mathcal{S}$). This assumption is not required to ensure the termination of the OTR algorithm, but it makes it possible to terminate faster in some cases: as soon as node u decides, then if u is in a "good period" its decision can be transmitted to all other nodes in $\mathcal{V}_\mathcal{S}$. As observed in [101], assuming good periods in an asynchronous system is often more realistic than assuming that the system is partially synchronous.

6.4 Solving Consensus with the OTR Algorithm

This section presents how to solve consensus in the context of OppNets. The solution presented here is based on a variant of the One-Third-Rule (OTR) Heard-Of algorithm proposed in [102], extending it with a delay-tolerant content-based communication middleware for accommodating the design requirements of OppNets.

6.4.1 Overview of the Heard-Of Model

A computation in the *Heard-Of* (HO) model [102] evolves in asynchronous communication-closed rounds, without any need for a failure detector. In each round, each process sends a message to all the other processes and then waits to receive similar messages sent in the same round. Late messages pertaining to former rounds are discarded. The features of a specific system are captured by a *communication predicate*, which is expressed in terms of *Heard-Of sets*: $HO(p, r)$ represents the set of processes from which process p "hears of" (i.e., receives some messages) at round r. A consensus problem is solved in the HO model by a *Heard-Of machine* defined by a pair $M = (A, \mathcal{P})$ where A is an algorithm and $\mathcal{P}$ is a communication predicate.

Several consensus algorithms have been expressed in the HO model [102]. These algorithms can hardly tolerate message loss, except for the *Paxos/LastVoting* (P/LV) algorithm and the *One-Third Rule* (OTR) algorithm. P/LV is a coordinated algorithm in which consensus can be solved by resorting to a coordinator (i.e., a server), whereas OTR is a non-coordinated algorithm. The OTR is a perfect candidate for opportunistic computing.

6.4.2 Overview of the OTR Algorithm

In [102], the OTR algorithm is defined with the formalism of the HO model, but similar structure and decision conditions can be observed in other algorithms (*e.g.*, first round in [104]). Each round r in the OTR algorithm consists of two steps (see Algorithm 6.1): a sending step S_p^r in which process p sends its current contribution (for round r) to the other processes (line 3), followed by a transition step T_p^r in which, provided it has received enough contributions from the other processes (line 5), process p either takes a decision (line 8) or determines its contribution for the next round (line 6) and proceeds to that round (line 7).

Algorithm 6.1 The *One-Third Rule* algorithm

Initialization:
1: $x_p \leftarrow v_p$ $\{v_p$ is the initial value of process $p\}$

Round r:
2: S_p^r:
3: send <x_p> to all other processes
4: T_p^r:
5: **if** | *HO(p,r)* | > *2n/3* values **then**
6: $\quad x_p \leftarrow$ *the smallest most often received value*
7: $\quad$ **if** *more than* $2n/3$ *values received are equal to* $\overline{x}$ **then**
8: $\quad\quad$ DECIDE($\overline{x}$)

A node p can tolerate not to receive messages from up to one-third of the other participants in round r, while still being able to decide or proceed to the next round. In practice, in an OppNet this may occur because some of the other participants have not reached round r yet (for example because these participants are currently in suspend mode), or because some participants have indeed sent their contributions for round r but these messages have not reached node p yet (and possibly never will).

Moreover, with the OTR algorithm a consensus computation involving n participants can progress from round r to the next if *at least one* participant can receive contributions (pertaining to round r) from more than $\frac{2n}{3}$ other participants. In an OppNet where message delivery can sometimes be delayed significantly (at least for some receivers), this property of the OTR algorithm is an asset, for the consensus computation can proceed from one round to the next as soon as one node has received enough contributions to do so.

Conversely, at least $\frac{2n}{3}$ participants must send contributions in each round, since this is a requirement for the algorithm to proceed to the next round or to the final decision.

Based on these observations the assumptions made in the system model (Section 6.3) can be refined so as to account for the specific requirements of the OTR algorithm:

- At each round r, the number of participants sending their contribution in the network should not be smaller than $\frac{2n}{3}$ (which means that about $\frac{n}{3}$ participants can actually "skip" a round without preventing the computation to progress). By extension, the number of crash failures among the participants must be smaller than $\frac{n}{3}$.
- At each round r, message loss should be such that *at least one* node can receive more than $\frac{2n}{3}$ contributions (which means that receive omissions are admitted for most

of the participants but one, which should be able to receive enough contributions to proceed to the next round). Using the HO formalism, this assumption can be expressed as:

$$\forall r,\, \exists\, p \in \mathcal{V}_s\, s.t.\, |HO(p,r)| > 2n/3$$

Note that these requirements fit perfectly with the characteristics of an OppNet, in which node availability and message delivery cannot be guaranteed. Moreover, the *n-to-n* communication pattern used in the OTR algorithm is satisfied by the epidemic routing model, since with this model sending a message to many or all nodes is not really different from sending a message to a single node.

According to [102] the communication predicate $\mathcal{P}_{(\mathcal{C}_0)\infty}$ associated with the OTR algorithm ensures that the consensus can be solved if there exists a round r_0 where $\mathcal{C}_0$ holds:

$$\exists\, \Pi_0\, s.t.\, |\Pi_0| > 2n/3,\, \forall p \in \mathcal{V}_s :\, HO(p, r_0) = \Pi_0$$

In other words, during r_0 all participants must be able to receive contributions from the very same subset of more than $\frac{2n}{3}$ participants (with $n = |\mathcal{V}_s|$), so they can make the same contribution. This predicate can be expressed nicely in terms of *HO sets*, and it is sufficient to ensure that the consensus is solved. Yet it does not define a necessary condition. If there exists a round r_0 where the contributions collected by participants are such that the smallest most frequent value is the same for all participants, then the consensus can be solved as well, even though all participants may not have received these values from the same contributors and even if the other values (i.e., other than the smallest value) are not the same for all participants. This condition can hardly be expressed in terms of *HO sets*, yet it is weaker than predicate $\mathcal{P}_{(\mathcal{C}_0)\infty}$ and allows more flexibility in the system model.

6.5 Opportunistic Implementation of the OTR Algorithm

ADAM (*Agreement in Disconnected Adhoc Mobile networks*) is the consensus system designed along this chapter. Its general architecture is composed of two layers: the lower layer is based on DoDWAN protocol, and the upper layer is an implementation of the OTR algorithm that interfaces with the communication layer.

6.5.1 Communication Abstraction Layer

Currently, the communication layer is based on DoDWAN presented in Chapter 3. This section gives an overview of a communication abstraction layer for leveraging the *publish/subscribe* application programming interface (API) provided by DoDWAN. The main aim of this layer is to allow users to deal with high-level information (e.g., group identifier) which is then used to create the corresponding communication-level information needed by DoDWAN (e.g., descriptors and payloads). By doing so, this abstraction layer can ease the construction of a consensus algorithm which is more *readable,* more structured and therefore more intuitive, as presented in the next section.

This layer provides an API, presented in Code 6.2. With this API, the content-driven nature of message selection is based on the notion of group. A group, identified by a group identifier *(grpId)*, is a set of nodes that cooperate in a common task and are thus potentially interested by the same kind of messages[2]. The *interest profile* of a node is a compilation of the ids of all the groups it belongs to. Function *subscribe* (line 2) allows a process to specify that it is interested in receiving the messages for a given group. This function requires DoDWAN (by providing it with the corresponding descriptor) to add the specified group id in the node's interest profile, and consequently the node will try to collect messages addressed to this group from any neighbor it will meet thereafter. Whenever a message is received that matches a group the node belongs to, a *receive* event is triggered (line 3) accordingly.

Code 6.2 The communication abstraction layer API

```
1: Function publish (msgId, grpId, sndId, BODY, [dln])
2: Function subscribe (grpId)
3: Event receive (msgId, grpId, sndId, BODY)
4: Function cancel (msgId)
5: Function relay (grpId)
```

Sending a message in the network is done with the *publish* function (line 1), that takes as parameters identifiers for the message itself *(msgId)*, for its sender *(sndId)*, and for the group of nodes it is addressed to *(grpId)*. These parameters are used to provide DoDWAN with the corresponding descriptor. The body of the message is just perceived as a payload by DoDWAN.

Each message can optionally be assigned a set lifetime when it is published (last parameter in line 1). The dissemination of a message can also be canceled explicitly on a mobile node (line 4).

A node can be configured so as to serve as a benevolent carrier for messages addressed to a group it does not belong to. With the API this is obtained through the *relay* function (line 5), which basically has the same effect as function *subscribe*, except that messages received by a benevolent carrier will not trigger any *receive* event on that node.

6.5.2 Opportunistic OTR Algorithm

The opportunistic implementation of the OTR algorithm based on the opportunistic communication layer is shown in Algorithm 6.3. A consensus session is initiated by calling function *startSession*, taking the initial value for the local participant (line 4), the group identifier (line 5), and the number of participants in this group (line 6) as parameters. A subscription is then set for the group (line 8) before starting the first round of the OTR algorithm (line 9).

At each round, the current contribution is published (line 12), and the identifier of the message hence published is recorded in *contribIds* so it can be canceled later (line 13).

When a contribution is received from another participant, the behavior of the receiver depends on whether this contribution pertains to a new round (lines 18-23), to the current round (lines 25-34), or to a former round (line 36).

In the first case the receiver cancels messages pertaining to the current round (line 18-19), adopts the received contribution (line 20) before moving to the new round (line 23).

[2]For the upper consensus layer, a group is the subset $\mathcal{V}_S$ of nodes involved in a consensus.

Algorithm 6.3 Opportunistic version of the OTR algorithm

```
Initialization:
id_p ← id of local node                                   {must be unique in the network}
contrib_p ← {}                                            {contributions received for r_p (multi-set)}
contribIds_p ← {}                                         {ids of messages received during r_p (set)}

Function startSession(grpId, nbNodes, v):
x_p ← v                                                   {initial value for node p}
grpId_p ← grpId
nbNodes_p ← nbNodes
solved ← false
subscribe(grpId_p)
startRound(1)

Function startRound(r):
r_p ← r
msgId ← genId()                                           {call id generator}
publish(msgId, grpId_p, id_p, CONTRIB(r_p, x_p))
contribIds_p ← contribIds_p ∪ {msgId}

Upon receive (msgId, grpId, sndId, CONTRIB(r, x)) do
if solved then
    cancel(msgId)
switch (r)
case (r > r_p):
    for id ∈ contribIds do
        cancel(id)
    x_p ← x
    contrib_p ← {x}
    contribIds_p ← {msgId}
    startRound(r)
case (r = r_p):
    contrib_p ← contrib_p ∪ {x}
    contribIds_p ← contribIds_p ∪ {msgId}
    if |contrib_p| > 2/3 * nbNodes_p then
        x_p ← smallest most often received value in contrib_p
        if more than 2n/3 values in contrib_p are equal to x_p then
            decide(x_p)
            msgId ← genId(grpId)
            publish(msgId, grpId_p, id_p, DECISION(x_p))
        else
            startRound(r_p + 1)
case (r < r_p):
    cancel(msgId)

Function decide(v):
if ¬solved then
    solved ← true
    x_p ← v
    for id ∈ contribIds do
        cancel(id)
    contribIds_p ← {}

Upon receive (msgId, grpId, sndId, DECISION(v)) do
decide(v)
```

In the second case it checks if enough contributions have been received to either make a decision (line 30) or start the next round (line 34). When a decision is made locally (line 30), it is published (line 32) with a unique message identifier, that is produced using *grpId* as a seed (line 31). Thus, if several nodes make a decision in the same session, all messages carrying this decision will have the same identifier and will thus be considered as duplicates of the same message by the communication layer. As explained in Section 6.3, publishing the decision is not required by the OTR algorithm, but it can help terminate a consensus session faster.

In the third case it simply discards the contribution it has just received, but cancels the corresponding message so as not to take part in its dissemination (line 36). Note that message cancellation is here used systematically as a means to prevent messages that pertain to former rounds to keep disseminating in the network. This form of cross-layering between the OTR algorithm and the opportunistic communication layer helps reduce the cost of epidemic message dissemination.

Function *decide* is called when a decision is made locally (line 30), or when a decision message is received from another participant (line 43). In any case this function is run only once on each node (line 37): all pending messages are canceled (lines 40-41).

6.6 Experimental Evaluation

ADAM has been fully implemented, and validated in real conditions using a small flotilla of smartphones as mobile nodes. During this experiment it would have been most interesting to compare ADAM against other consensus systems. Unfortunately, ADAM is currently the only consensus system for OppNets whose implementation is openly available for application developers, for its source code is distributed under the terms of the GNU General Public License[3]. It would be possible to implement the Paxos/Last Voting (P/LV) algorithm, which is well-defined in [101], so as to compare its performance against ADAM in real conditions. However, this coordinator-based consensus algorithm requires temporaneous end-to-end connectivity between the coordinator (i.e., the server) and all the other processes in the network. Hence, this algorithm cannot work in OppNets.

6.6.1 Experimentation Conditions

In order to demonstrate that ADAM can indeed solve consensus in an OppNet, volunteers have been equipped with HTC smartphones, whose Wi-Fi chipsets were configured to operate in ad hoc mode. Each smartphone ran a small Android application (named *iAgree*) based on ADAM. This simple application, whose user interface is shown in Figure 6.1, allows a user to initiate new consensus sessions, and to join, participate, and display the status of ongoing and past sessions. The volunteers were asked to carry their smartphone while roaming the laboratory building or its surroundings, and to use application *iAgree* every now and then. Trace logs were collected and analyzed after the end of the experiment.

[3]`http://www-casa.irisa.fr/adam/`

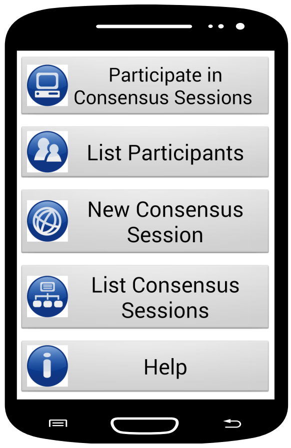


Figure 6.1: iAgree application

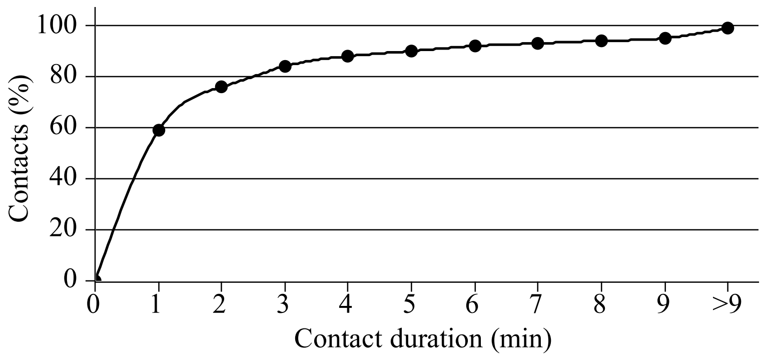


Figure 6.2: Cumulative distribution of radio contact durations

The experiment spanned over 8 hours and involved a small population of 7 volunteers (and as many smartphones). During this experiment ADAM was configured to give each message a lifetime of 12 hours. No message was therefore removed because of an exhausted lifetime. Moreover, because only 7 smartphones were available for this experiment, none of them was configured to serve as a benevolent carrier: every smartphone was systematically enrolled as a consensus participant.

6.6.2 Results

3424 radio contacts occurred between smartphones during this 8 hour experiment, with an average contact duration of 162 seconds. The cumulative distribution of contact durations is presented in Figure 6.2. It can be observed that almost 60% of radio contacts lasted for less than a minute, which confirms the transient nature of the radio contacts established between smartphones.

A timeline of the evolution of the average number of neighbors is presented in Figure 6.3, and the cumulative distribution of this number is shown in Figure 6.4. It can be observed that each smartphone had at most one neighbor during about 40% of the experiment's duration, but was actually alone (*i.e.*, with no neighbor) during more than

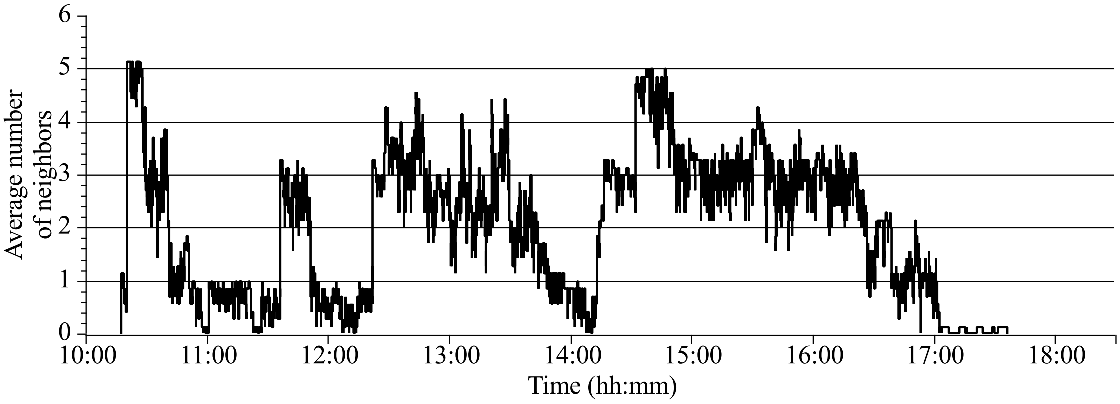


Figure 6.3: Timeline of the average number of neighbors during the experiment

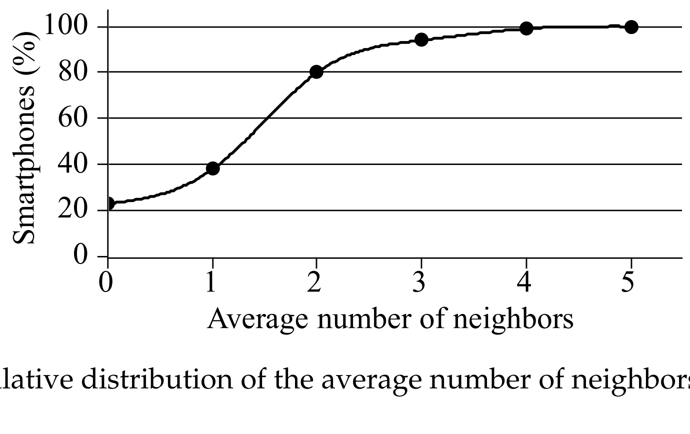


Figure 6.4: Cumulative distribution of the average number of neighbors perceived by all smartphones

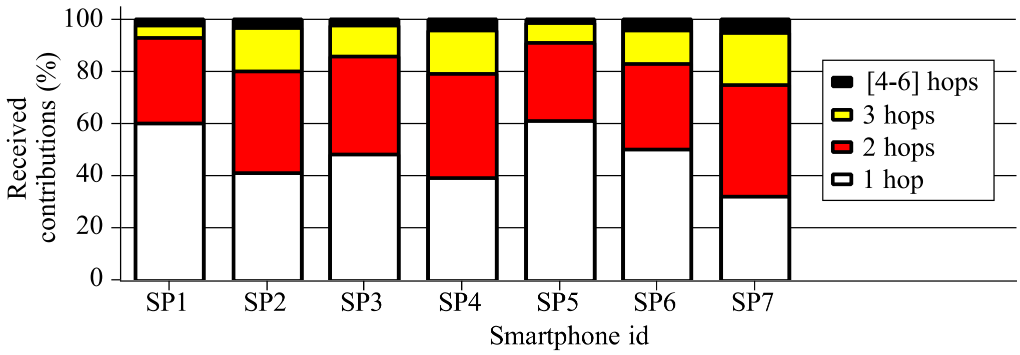


Figure 6.5: Amount of contributions received in a direct/multi-hop manner per smartphone

20% of that time. Moreover, there was no period of time during which all smartphones were connected all together (each smartphone would otherwise have detected 6 neighbors simultaneously). These results confirm the dynamic and disconnected nature of the network, whose topology changed continuously and rapidly during the whole experiment.

151 consensus sessions were initiated by users (either sequentially or concurrently) during the experiment, and running these sessions led to the exchange of 4530 contributions among the smartphones.

Figure 6.5 shows the distribution of the number of hops required for contributions to reach each smartphone. For example, 50% of the contributions received by smartphone *SP6* were received directly from the sender (1-hop), 33% contributions were received by *SP6* after 2 hops, etc. The smartphones actually received most of the contributions after multi-hop trips during the experiment, a few of these trips requiring up to 6 hops between sender and receiver. This observation confirms the interest of multi-hop relaying between smartphones.

Figure 6.6 illustrates how one particular consensus contribution sent by smartphone *SP2* actually disseminated during the experiment. A few minutes after this particular contribution was published (*i.e.* sent to all other nodes) by ADAM on *SP2*, a radio contact was established with *SP6*, which thus got a copy of the contribution and became a new carrier for this contribution. *SP2* later managed to forward the contribution to *SP7*, while *SP6* forwarded it to *SP1* first, and later to *SP3*. The contribution thus kept disseminating, until it reached the last smartphone *SP5*, almost half an hour after it was initially published on *SP2*. This observation demonstrates the interest of the *store, carry and forward* principle in an OppNet such as that formed by the smartphones during the experiment.

Since consensus contributions had to propagate opportunistically between the smartphones, each consensus session could take a while before a decision was made. Actually, 9% of the sessions could not reach a decision during the experiment, but these sessions were initiated shortly before the end of the experimentation period. Figure 6.7 presents the cumulative distributed function of the execution times for all completed sessions. The execution time of a session is here defined as the time elapsed between the sending of the

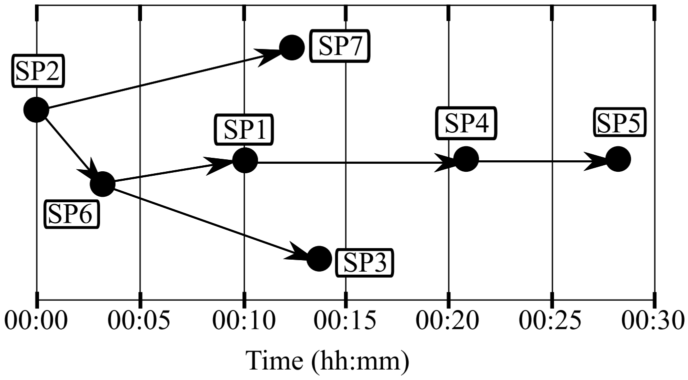


Figure 6.6: Timeline of the dissemination of a contribution during the experiment

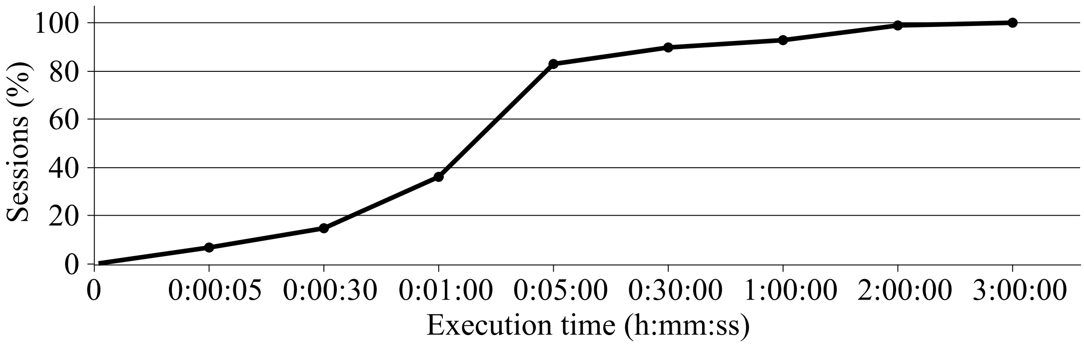


Figure 6.7: Cumulative distribution of the execution times of sessions

first contribution by one of the participants, and the last decision made or received by any of the participants. It can be observed that a few sessions were completed in only a few seconds, but most of them required several minutes, and some even took a couple of hours to complete. These results clearly show that consensus solving in an OppNet can indeed take a while, but is perfectly feasible provided an appropriate communication model is used.

Figure 6.8 shows the distribution of the number of rounds required to solve consensus in all completed sessions. It can be observed that consensus was obtained in only one round for about 5% of the sessions. Furthermore, most sessions required a couple of rounds to complete.

Although the real-life performance evaluations of JOMS, JION and ADAM were all performed in similar scenarios using a small flotilla of smartphones as mobile nodes, the obtained results were different for each experiment. This confirms the fact that conducting a real-world performance evaluation of a middleware system in an OppNet can accurately capture system interactions and environment characteristics. However, real-world performance evaluation imposes logistical and practical limitations on the experiment's repeatability. Simulations provide advantages in terms of repeatability, but they also commonly fail to take into account system interactions, and can only provide limited fidelity, especially for wireless related research.

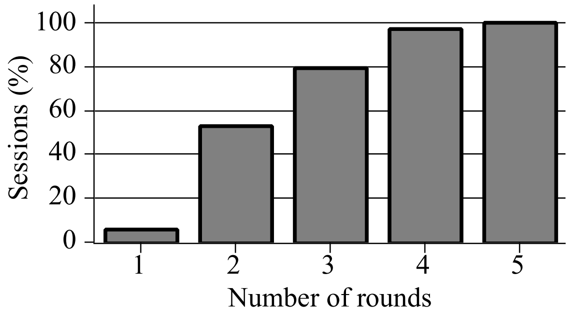


Figure 6.8: Cumulative distribution of the number of rounds required to solve consensus

6.7 Simulation

This section presents an evaluation of the performance of ADAM using simulation.

6.7.1 Simulation: Benefits and Challenges

The experiment presented in Section 6.6 was mostly meant to serve as a proof-of-concept rather than as an exhaustive investigation, to demonstrate that ADAM can indeed operate satisfactorily in an OppNet.

Larger experiments would notably make it possible to observe phenomena that could not be highlighted with a flotilla of only 7 smartphones, such as the effect of relying on mules (i.e., mobile nodes that carry messages they are not directly interested in) to carry consensus contributions, or the effect of receive omissions when consensus sessions involve many participants.

Actually, organizing such larger experiments would notably require buying hundreds of smartphones and distribute these smartphones to as many volunteers. It is worth restating that volunteers cannot simply use their own smartphones since they must be rooted in order to operate in ad hoc mode, as explained in Section 4.7. Furthermore, extending experiments to several weeks is not as simple as it seems. Such long-term experiments have already been conducted in IRISA laboratory [105], and it turned out that extending the scope (in terms of population, duration, and geographical area) of an experiment does not necessarily provide more significant results. In fact these results are often quite frustrating because the enthusiasm of volunteers often declines rapidly over a couple of days: the smartphones get forgotten in a drawer, or their battery drains out and is never recharged.

In order to provide a comprehensive evaluation of an OppNet-dedicated system, researchers usually rely on simulation tools. Simulators allow to design and test viable solutions also in the absence of real prototypes. Yet any parameter in a simulation (e.g. mobility model, radio propagation model, application-level traffic induced by the use-case scenario, etc.) is debatable unless it is derived from real-world experimentation. This is the reason why our approach consists in analyzing real-world experiments in order to get inputs (about radio contacts, about user behavior, etc.) that can later be used in simulation models.

6.7.2 MobSim

In order to evaluate the performance of ADAM in a simulated OppNet, ADAM is interfaced with MobSim, a network simulator designed and implemented in our laboratory.

MobSim allows to investigate the performance of disruption-tolerant systems in a simulated OppNet. It provides a customizable simulation area along with several mobility models, e.g., the Random Waypoint (RWP) mobility model and the Truncated Levy Walk (TLW) mobility model. Contrary to more commonly used wireless network simulators MobSim allows developers to run their actual code directly on every network node, hence taking into account not only algorithmic issues but all the implementation choices as well.

MobSim allows nodes to intercommunicate using a *hub*. Indeed, each node is considered as an independent process by MobSim, and the hub itself is also considered as another independent process. Processes are connected with the hub via TCP or UDP connections.

The hub can be associated with a simulation environment, that simulates the mobility of nodes and their interactions according to their actual positions.

When MobSim is started, it initiates the hub, that starts listening for messages coming from mobile nodes. When it receives a message from a new node, this message actually informs the hub about the existence of this node. The hub then requests the simulation environment to start simulating the mobility of this node. Using the simulation environment, the hub is able to know the neighbors of each node according to its actual position, so as to manage message forwarding between nodes.

When no new messages are received from a known node for a specific period of time (*timeout*), the hub can decide to forget this node and require the simulation environment to delete it permanently.

MobSim simulates the states of nodes: "online" and "offline" states. When a node switches to offline state, the hub informs the simulation environment to continue simulating its mobility while preventing it from communicating with other nodes (that is, it informs the simulation environment to prevent this offline node from communicating with neighbors).

The simulation environment can be also associated with a visualizer so as to visualize its operations (e.g., adding a new node, moving a node, etc.). Furthermore, the simulation environment can record a log about the evolution of a given simulation, granting developers the ability to replay this simulation in order to perform further tasks.

Levy Walk Mobility Model For our simulation, MobSim was configured to use the Levy Walk mobility model, which shows similar statistical characteristics as human walk [106]. Indeed, Rhee et al. analyzed a set of real-life movement traces collected from users across different cities [106]. Consequently, they concluded that human movement can be modeled using Levy Walk. This section provides a brief background about this mobility model.

According to the Levy Walk mobility model, a "walker" moves by making a sequence of *steps*. Each step combines a *flight and a pause,* and is represented by a tuple

$S = (l, \theta, \triangle t_f, \triangle t_p)$. Here:

1. $l > 0$ denotes the *flight length*, whose distribution $p(l)$ follows the Levy distribution: $p(l) \sim 1/l^{(1+\alpha)}$, where $\alpha \in]0, 2[$. The smaller the value of α, the longer the *flight lengths* performed by the walker. The value of a *flight length* cannot exceed a truncation factor, called τ_l.

2. θ denotes the direction of the flight. It takes a value from a uniform distribution of angle within $[0^{\circ}, 360^{\circ}]$.

3. $\triangle t_f > 0$ denotes the *flight time,* that is modeled as: $\triangle t_f = k * l^{1-\rho}$, $0 \leq \rho \leq 1$ where k and ρ are constants. When $\rho = 0$, flight times are proportional to flight lengths. When $\rho = 1$, flight times are constant.

4. $\triangle t_p > 0$ denotes the *pause time,* whose distribution $\psi(\triangle t_p)$ follows the Levy distribution: $\psi(\triangle t_p) \sim 1/\triangle t_p^{(1+\beta)}$, where $\beta \in]0, 2[$. The smaller the value of β, the longer the *pause times* taken the walker. The value of a *pause time* is always truncated by τ_p and cannot exceed this value.

Based on this model, a walker moves in a given area in the following manner:

1. The walker picks up its initial location randomly in the area.

2. At every step, it generates an instance of tuple $(l, \theta, \triangle t_f, \triangle t_p)$ randomly from their corresponding distribution. If *flight length* or *flight time* is negative, or greater than their corresponding truncation factors, then this step is immediately discarded and a new step is regenerated.

3. The walker performs the flight and waits for $\triangle t_f$ after reaching the destination.

4. This process is repeated after the step time (i.e, *flight time+pause time*), until the end of simulation.

More details can be found in [106].

6.7.3 Simulation Setup

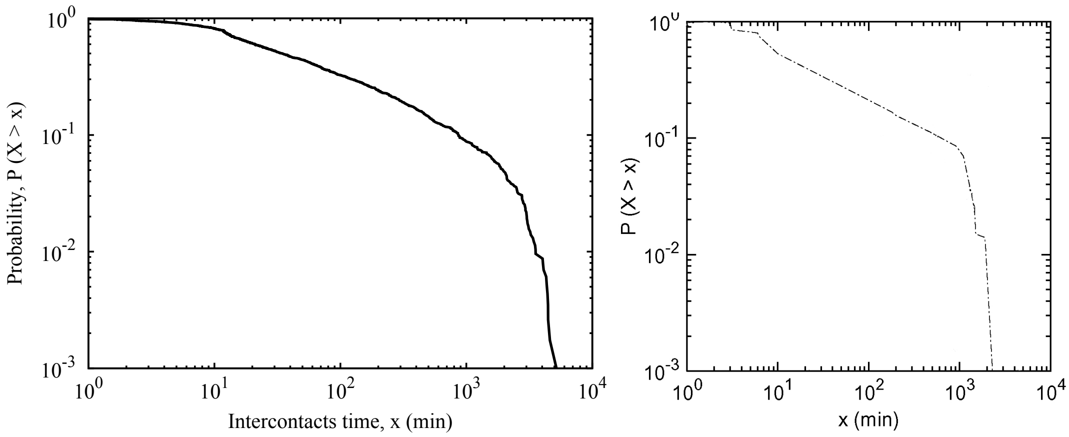
Levy Walk Setup Before starting to evaluate the performance of ADAM using MobSim, one must first find the set of parameters that enables the Levy Walk mobility model to show similar statistical characteristics as a real-life scenario.

To do so, we analyzed the real-world experiment performed during the IEEE INFOCOM 2005 conference where 41 wireless devices were carried by attendees for 3 to 4 days [107]. The Levy Walk mobility model was then configured to simulate the INFOCOM conference using the set of parameters presented in [106]. These parameters are shown in Table 6.1.

According to Rhee et al., *in DTNs, [...] the time duration between two consecutive contacts of the same two nodes (called intercontact time) is an important metric.*

Parameter's Name	Symbol	Value
Simulation area		1.5 x 1.5 km^2
Simulation duration		4 days
Communication radius		50 m
Number of nodes		40 nodes
Flight length Levy coefficient	α	1.8
Pause time Levy coefficient	β	1.8
Truncation flight	τ_l	200 m
Truncation pause	τ_p	1 h
Flight time parameters	k, ρ	$k = 30.55$ and $\rho = 0.89$, when $l < 500m$ $k = 0.76$ and $\rho = 0.28$, when $l \geqslant 500m$

Table 6.1: Parameters of the Levy Walk mobility model



(a) ICT distributions obtained using Levy Walk

(b) ICT distributions as observed in INFOCOM

Figure 6.9: Comparison between the ICT distributions obtained using Levy Walk with the ICT distributions observed in INFOCOM (log scale)

Therefore, we compared the Intercontact Time (ICT) distribution[4] generated by Levy Walk (Figure 6.9a), with the ICT distribution observed during INFOCOM (Figure 6.9b). It can be observed that Levy Walk can recreate the ICT distribution of the INFOCOM conference.

These results prove that the set of parameters shown in Table 6.1 allows the Levy Walk mobility model to show similar statistical characteristics as the real-life experiment of INFOCOM conference.

MobSim Setup MobSim was configured to simulate an area of $400 \times 200\, m^2$. Nodes in the simulation area represented human beings equipped with Wi-Fi-capable handheld devices, assuming a wireless range of 30 m. Nodes were moving according to the Levy Walk mobility model with the set of parameters presented in Table 6.1.

[4]The ICT distribution is calculated as the complementary cumulative distribution function (CCDF) of intercontact time.

Parameter's Name	Symbol	Value
Simulation area		400 x 200 m^2
Simulation duration		8 hours
Communication radius		30 m
Number of participants	$\lvert\mathcal{V}_{\mathcal{S}}\rvert$	40 participants
Number of mules	$\lvert\mathcal{V}_{\mathcal{M}(\mathcal{S})}\rvert$	0 mules (1st run) 50 mules (2nd run) 100 mules (3rd run) 150 mules (4th run)
Session lifetime	$\mathcal{T}_{\mathcal{S}}$	4.5 hours (1st run) 2 hours (2nd, 3rd and 4th runs)
Flight length Levy coefficient	α	1.8
Pause time Levy coefficient	β	1.8
Truncation flight	τ_l	200 m
Truncation pause	τ_p	1 h
Flight time parameters	k, ρ	$k = 30.55$ and $\rho = 0.89$, when $l < 500m$ $k = 0.76$ and $\rho = 0.28$, when $l \geqslant 500m$

Table 6.2: Simulation parameters

Four different simulation runs were carried out in order to evaluate the performance of ADAM. Each simulation run lasted for 8 hours. The simulation area was populated with 40 participants, and the number of mules was different in each simulation run: there was no mule for the first simulation run, 50 mules for the second one, 100 mules for the third one and 150 mules for the fourth one. 40 consensus sessions were initiated in each simulation run. The lifetime of consensus sessions was 4.5 hours during the first simulation run (i.e., the simulation run without mule) and 2 hours during the three other simulation runs (i.e., the simulation runs with mules). These simulation parameters are summarized in Table 6.2.

6.7.4 Simulation Results

The contact durations distribution and the ICT distribution between participants were the same in the four simulation runs. Indeed, participants in the four simulation runs were all moving using the Levy Walk mobility model using the same set of parameters. The results of the cumulative distribution of contact durations and the ICT distribution between participants are shown in Figure 6.10 and 6.11, respectively. It can be observed that almost 50% of radio contacts lasted for less than 5 minutes (point A in Figure 6.10), which confirms the transient nature of the radio contacts established between mobile devices. Furthermore, almost 50% of ICT was about 18 minutes(point A in Figure 6.11), which confirms the interests of having mules serve as benevolent message carriers between these participants in order to accelerate consensus solving.

The 40 consensus sessions that have been initiated in each simulation run, have all been completed and their final decisions have reached all the participants. Figure 6.12a and 6.12b present the cumulative distribution of the execution times of all sessions during the first simulation run (i.e., without mules) and during the other runs (i.e., with mules),

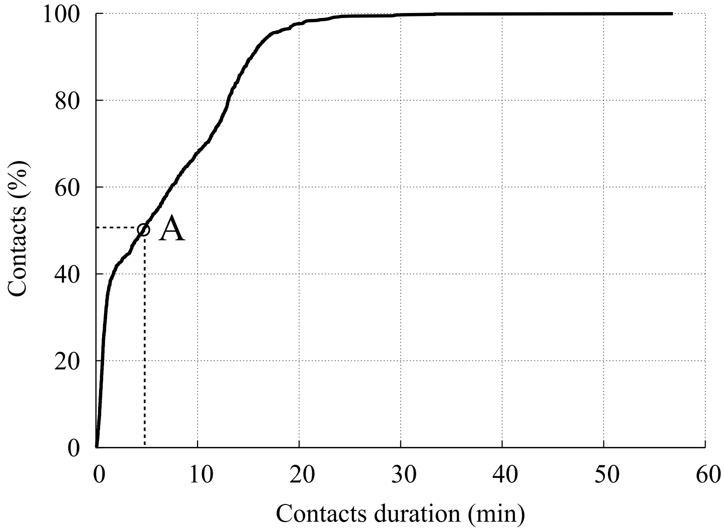


Figure 6.10: Cumulative distribution of contact durations between participants

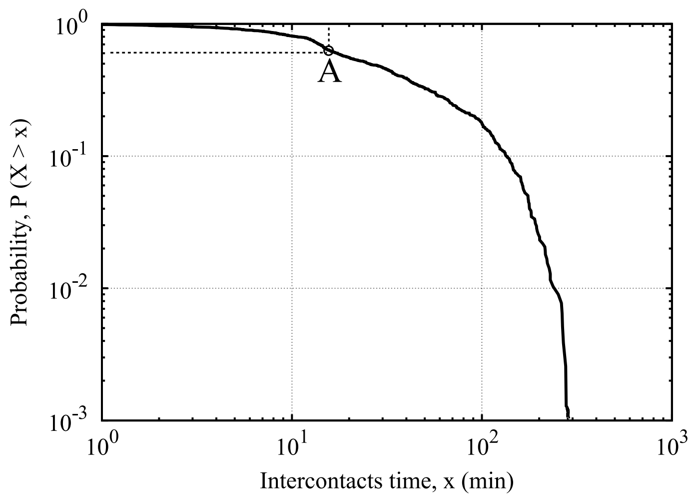
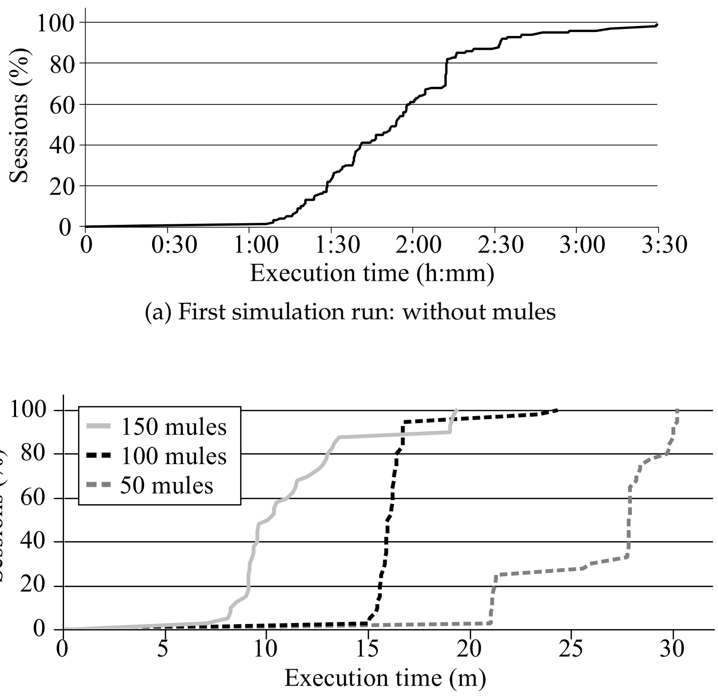


Figure 6.11: ICT distribution between participants (log scale)



(a) First simulation run: without mules

(b) Second, third and fourth simulation runs: with mules

Figure 6.12: Cumulative distribution of the execution times of sessions during the simulation

respectively. These results confirm again the interest of having mules in an OppNet to accelerate consensus solving. Indeed, when using mules all sessions have been solved in less than 30 minutes, while it took at most an hour and up to 3 hours to solve consensus when using no mules. Figure 6.12b shows that, in general, the greater the number of mules in a session, the faster the session is solved. Furthermore, the ICT distribution of nodes plays an important role on the execution time of sessions: the less the ICT between nodes in an OppNet, the faster the sessions are solved in this network.

The cumulative distribution of the number of rounds required to solve consensus in all sessions during the different simulation runs is shown in Figure 6.13. All sessions required at least 2 rounds to complete, but none of them required more than 5 rounds, though. Actually, about 80% of the sessions were completed in 3 rounds or less. Interestingly, these results are coherent to those observed during the real-life experiment, as shown in Figure 6.8.

As mentioned in Section 6.5, the opportunistic OTR algorithm can progress from a round r to another round (r'>r) either by receiving enough contributions that pertain to r (2/3 progress rule) or by receiving a contribution that pertains to r' (future progress rule). Figure 6.14 shows the distribution of the rules that triggered round progression during the simulations. It can be observed that the "future progress rule" was the major trigger. These results confirm the importance of handling contributions that pertain to future rounds (lines 20-26 in Algorithm 6.3) in speeding up the execution of consensus sessions by reducing “useless” wait time.

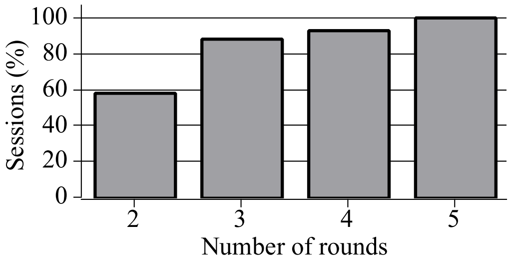


Figure 6.13: Cumulative distribution of the number of rounds required to solve consensus

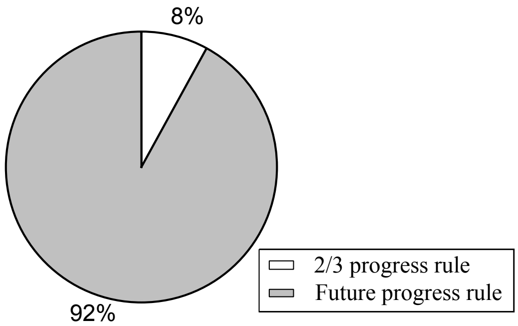


Figure 6.14: Distribution of the ways by which the opportunistic OTR algorithm progressed from a round to another

6.8 Discussion

The above-mentioned results globally demonstrate that ADAM, which combines a variant of the OTR algorithm with an opportunistic communication layer, is effective at solving consensus in an OppNet.

Leveraging ADAM by the middleware systems presented in the former sections (i.e., JOMS and JION) can have both favorable and unfavorable consequences.

On the one hand, leveraging ADAM would allow these systems to support their operations differently so as to be well-adapted for OppNets. For example, JION can leverage ADAM to support the tuple space programming model differently by which the *write* operations are disseminated network-wide and a *take* operation for a given entry calls for a consensus session to agree to delete the entry network-wide. The same technique could be used by JOMS so as to support the message queue programming model in a distributed server-less manner without leveraging the concept of queue manager (QM). ADAM would allow these middleware systems to set their own configurations (e.g., where to store a queue manager in JOMS). ADAM would also facilitate the development of self-adaptive JMS and JavaSpaces applications, that can agree to perform some adaptations based on some criteria (e.g., migrating a queue manager between mobile devices in JOMS when resources are not sufficient).

On the other hand, leveraging ADAM by other middleware systems might cause some inconvenience. Indeed, the experimental and simulation results clearly show that consensus solving in an OppNet can indeed take a while to complete. Therefore, performing operations that leverage ADAM might be time-consuming. However, the performance gain achieved through consensus could sometimes compensate such extra cost.

More investigation is required to evaluate the cost/performance ratio of JOMS and JION once leveraging ADAM.

6.9 Enhancing the Performance of ADAM

ADAM could be augmented with some mechanisms in order to enhance its performance in the context of OppNets. This section describes in detail these mechanisms which represent interesting perspectives for future work.

Preliminary Rounds Mechanism The first perspective consists in speeding up the execution of the opportunistic variant of the OTR algorithm, so as to reduce its computation load. In general, the earlier the algorithm achieves consensus, the less resources (e.g., bandwidth and memory) are consumed.

To do so, some *preliminary rounds* could be added prior to the OTR algorithm so as to allow the convergence of the initial values of participants before starting to solve consensus. During the additional preliminary rounds, each participant exchanges its initial value with the directly-connected participants. The involved participants then compute a new value using a given convergence function (e.g, minimum value) so as to adapt the obtained value for the next exchange. The total duration of the preliminary rounds could be bounded by the system. At the end of the preliminary rounds, the diversity of initial values is in fact diminished. Henceforth, participants can start solving consensus normally using the opportunistic variant of the OTR algorithm presented in Section 6.5.

The mechanism of preliminary rounds takes advantage of the opportunistic contacts in an OppNet, in the sense that each contact is an opportunity for two participants to allow the convergence of their initial values before starting the OTR algorithm. By doing so, this mechanism reduces the diversity of initial values, and hence allows consensus sessions to be solved with fewer rounds.

Bootstrapping Mechanism Knowing the exact number of participants in a given session is essential to solve it. The current version of ADAM assumes that each participant knows in advance the sessions to which it belongs and the number of participants in each session. In other words, the definition and creation of sessions is assumed to be application-dependent. In this regard, ADAM could be augmented with a *bootstrapping mechanism*, that relieves the burden on application developers to deal with sessions creations. In fact, application developers need only to "create" consensus sessions, but do not really care about "how" the involved participants are informed, nor "which" information the participants receive to solve consensus.

When a node wants to act as a participant in an OppNet, it sends a declaration to all other nodes in the OppNet and waits for notifications about the creation of new consensus sessions.

Declarations are expressed as (*name*, *profile* and *lease*) tuples; where *name* denotes the identity of the participant, *profile* defines the types of consensus sessions in which the participant wants to participate and *lease* denotes the date until which the node acts a participant. Here, it is assumed that the types of consensus are application-level defined attributes, which are shared between nodes (using some directory services, for example). These types of consensus can be thought of as pre-defined groups to which participants could be associated: when trying to solve a consensus that belongs to a given group, only participants from this group can participate.

Nodes that receive declarations that pertain to new participants add the received information to a local *participants table*.

The node that starts a new session is called a *bootstrapper*, which must define three attributes: *session id* (*grpId*), *participants list* ($\mathcal{V}_S$) and *session lifetime* ($\mathcal{T}_S$). *Session id* is used to identify the session during which the consensus will be solved. It is important to make it unique, since it is used to distinguish between different consensus sessions in an OppNet. Therefore, when a session is created, the session id is appended with a local host's unique identifier (e.g., the IMEI on a smartphone or the MAC address for a local network interface). The bootstrapper, using *participant list*, leverages its local participant table to enumerate the participants that can participate to solve the consensus session. *Session lifetime* is meant to specify how long the participants should wait to solve the consensus, to prevent them from waiting forever.

Henceforth, ADAM leverages the underlying opportunistic communication layer in order to notify all the participants mentioned in *participant list* ($\mathcal{V}_S$) about the creation of a new session, in which they should participate. Once notified, the concerned participants start solving consensus by calling the function *startSession*, as mentioned in Section 6.3.

6.10 Related Work

Consensus problems have been studied extensively during the last decades, most often with system models that fit the characteristics of traditional wired networks. As a general rule, the papers assume that the network is static and connected, that is, any node can send a message at any time to any other node. Moreover, they also assume that the system can eventually become synchronous, or that it can be augmented with failure detectors, so as to go round the FLP impossibility result.

Most system models focus on node failures and tend to neglect link failures, though. According to Borran et al. [101] this bias may have its root in the FLP paper [103] (which assumes process crashes and reliable links), but solutions designed for environments where this bias is acceptable should not be used in environments where it is not acceptable.

Mobile ad hoc networks are such environments where reliable links should never be assumed, and for which system models must admit transient link failures. Yet several consensus protocols and algorithms have been proposed, based on overly optimistic assumptions.

Crash-tolerant broadcast protocols that can be used to solve consensus problems in mobile ad hoc networks are presented in [108]. These protocols assume the existence of oracles, that can predict contacts and transmission times between nodes at any point of time. Such an assumption can only be satisfied in very specific mobile networks such as those where the mobility patterns of nodes are planned or controlled explicitly.

A hierarchical cluster-based (HC) consensus protocol involving failure detectors is proposed in [109]. Mobile hosts are distributed into clusters, each cluster being controlled by a clusterhead. The protocol can tolerate faulty nodes, but links are assumed to be reliable.

An algorithm to solve Consensus with Unknown Participant or simply CUP is presented in [110]. This algorithm tries to solve consensus without knowing which nodes –and how many of them– are participating in the consensus, but the system model for this algorithm assumes reliable links and nodes that cannot crash. Variants of this algorithm, that try to solve Fault-Tolerant Consensus with Unknown Participants (FT-CUP), have later been proposed in [111] and [112]. Both variants can admit faulty nodes, but links are still assumed to be reliable.

An implementation of the Paxos/Last Voting (P/LV) algorithm is proposed in [101]. This round-based algorithm requires the election of a coordinator, which once elected collects the contributions of all other nodes until consensus is reached. Interestingly, unlike the abovementioned solutions the P/LV algorithm can admit both link and node failures, as it relies on the Heard-Of (HO) model that makes no distinction between both kinds of failures. It assumes end-to-end connectivity in the network, though, which is a reasonable assumption for traditional mobile ad hoc networks, but an unfit one for OppNets.

Byzantine agreement protocols for highly dynamic synchronous networks have been proposed in [113] and [114]. In [113] two randomized round-based protocols are presented, that can achieve almost-everywhere (AE) Byzantine agreement with high probability, even under a large number of Byzantine nodes and continuous adversarial churn. The network is represented as a sparse bounded degree expander graph, that is assumed to remain connected at any time, although its topology can change arbitrarily from round to round. The Neighbors On Watch (or simply NOW) protocol presented in [114] creates and maintains an expander overlay of clusters. Each cluster is used to inhibit the behavior of Byzantine nodes, and the overlay ensures communication among clusters. Although these protocols can support Byzantine failures, they can only run in connected networks and could hardly be used in OppNets.

With a closer view to details, Table 6.3 summarizes the comparison between the aforementioned consensus systems. This table presents whether a consensus system can tolerate node/link failures and whether the consensus system requires an end-to-end connection in order to solve consensus. Furthermore, it presents the programming model used in the consensus system (peer-to-peer or client/server) and whether the mobility pattern of hosts is planned or not. Finally, the table presents whether the consensus system has a real implementation, which is accessible to developers or it is only described in papers with no impact on the real world.

6.11 Summary

The One-Third Rule (OTR) algorithm is an elegant solution to solve consensus in networks where message loss can occur. Being based on the Heard-Of (HO) model, it is

Table 6.3: Comparison between the outlined consensus systems

Consensus System	Node failure tolerance	Link failure tolerance	No end-to-end connection	Peer-to-peer	Unplanned mobility	Openly distributed
Vollset et al. [108]	✓	✓	✓	✓	X	X
HC [109]	✓	X	X	X	✓	X
CUP [110]	X	X	X	X	✓	X
FT-CUP[111],[112]	✓	X	X	X	✓	X
P/LV [101]	✓	✓	X	X	✓	X
AE [113]	X	X	X	✓	✓	X
NOW [114]	✓	X	X	✓	✓	X
ADAM	✓	✓	✓	✓	✓	✓

well suited to support transient process and link faults, which makes it an ideal solution to solve consensus in OppNets. In such networks, messages propagate by being carried physically by mobile carriers whose mobility is usually neither planned nor controlled, so there is no guarantee that messages finally get delivered to their destinations.

This chapter presents ADAM, a consensus system that combines an implementation of a variant of the One-Third Rule (OTR) algorithm with a communication layer that supports network-wide, content-driven message dissemination based on controlled epidemic routing. Experimental results obtained with a small flotilla of smartphones along with larger experiments performed in simulated OppNets confirm that ADAM is effective at solving consensus problems in an OppNet. The source code of ADAM is now distributed under the terms of the GNU General Public License.

Part III

Conclusion and Future Works

7
Conclusion and Future Works

Contents

In Chapter 1, the objective of this work was presented: the main goal was to provide a set of high-level programming models in OppNets to facilitate the development of distributed applications over such networks.

The work reported in this dissertation focused on providing insight into the fundamental problems posed by OppNets, so as to analyze and solve the problems faced by application developers while dealing with these environments. The research focused on identifying well-known high-level programming models that can be satisfactorily implemented for OppNets, and that can prove useful for application developers. In order to demonstrate the feasibility of application development for OppNets, while assessing the benefits brought about by carefully designed middleware systems, a couple of such systems have been designed, implemented, and evaluated as part of this work.

In this chapter, we summarize the work presented in this work and we present some perspectives for future work.

7.1 Conclusion

OppNets have emerged as evolutions of legacy MANETs. The assumption in MANETs is the availability of a continuous end-to-end connection before any message exchange, requiring the sender and the receiver to stay simultaneously connected in a common island. Unfortunately, in real conditions, due to user mobility, limited power supply of their mobile devices and the intrinsic wireless link instability, the connectivity is such that an end-to-end path between mobile devices might never exist. OppNets leverage the *store-carry-and-forward* paradigm in order to turn one of the main obstacles of MANETs, namely the mobility, into a valuable asset by allowing mobile devices to carry messages with them while moving, until they encounter another device deemed more suitable to bring the message "closer" to the eventual destination.

Supporting networking in OppNets is only a first step, though. Opportunistic computing goes beyond the concept of opportunistic networking and provides a new paradigm to enable collaborative computing tasks.

However, designing and implementing distributed applications in the context of opportunistic computing is not a trivial task given the problems posed by OppNets. This dissertation focused on providing insight into these problems so as to identify the the main characteristics required in a programming model in order to be satisfactorily implemented in the context of opportunistic computing. By doing so, a set of major types of high-level programming models was identified as a basis for opportunistic computing. In order to demonstrate the feasibility of application development for OppNets, while assessing the benefits brought about by carefully designed middleware systems, a couple of such systems have been designed, implemented, and evaluated as part of this work.

A message-oriented middleware system was designed to provide two important programming models: the publish-subscribe programming model and the message queue programming model. This system, called *JOMS*, is actually a provider for the standard Java Message Service (JMS). Standard JMS applications using JMS message queues and topic can be directly deployed and executed in OppNets.

A fully-distributed, peer-to-peer coordination middleware system was also designed to provide the tuple space programming model. This system, called *JION*, is actually a JavaSpaces implementation with support for Future object. Standard JavaSpaces applications can be directly deployed and executed in OppNets.

A consensus middleware system was designed to provide consensus services. This system, called *ADAM*, is designed in order to enhance the performance of programming models that require some collaboratively-managed decisions.

The proposed middleware systems were supplemented with fully-functional implementations, that can be used in real settings, and that are all distributed under the terms of the GNU General Public License (GPL). Several measurement campaigns have been conducted, using either smartphones or netbooks, in order to evaluate the effectiveness and efficiency of these systems in real conditions. Furthermore, the performance of ADAM has also been evaluated in a simulated OppNet. The results obtained confirm the effectiveness and efficiency of the proposed implementations in an OppNet.

7.2 Perspectives for Future Work

This section is dedicated to identify and discuss some open research issues, so as to underline some interesting perspectives for future work.

7.2.1 Towards an Enhanced Set of High-level Programming Models

The set of programming models presented in this dissertation was identified as a *basis* for opportunistic computing. This set could be enhanced with other types of programming models, such as context-aware, transactional and mobile code programming models. The role of these new programming models is to introduce more flexibility into OppNets.

A context-aware programming model provides *context-aware computing*[115] in OppNets, so as to allow applications to be aware of the context in which they are being used. Context-aware applications could exploit the principal of *reflection* [116] to achieve dynamic adaptation to context changes. The role of reflection is to introduce more flexibility into context-aware applications.

A transactional programming model provides the concept of *transaction processing[117],* that allows application developers to handle a set of operations that can span across an OppNet. Using transactions, mobile devices can group their operations into a transaction, that acts as a single atomic operation. Either all operations within the transaction are performed, or none is.

A mobile code programming model provides the concept of *code mobility*[118], which allows the development of applications that can migrate from one mobile device to another in an OppNet while maintaining their execution state. This results in the ability to carry out a sequence of computations that span multiple mobile devices in the network. The ability of an application to migrate across mobile devices enables greater degrees of flexibility relative to traditional statically-installed code.

7.2.2 Leveraging Another Opportunistic Communication System

Although many communication protocols for OppNets have been proposed during the last decade, most of these protocols have only been described in papers as abstract algorithms, and tested using pseudo-code in simulators. Only a handful of these protocols have been actually implemented in middleware systems (and can thus be used in real conditions), but only a couple of these systems are openly distributed and are thus accessible to developers: DoDWAN, Haggle and DTN2.

The middleware systems presented in this dissertation currently rely on DoDWAN. Yet these middleware systems could theoretically be implemented above any other communication system in order to provide programming models in another kind of challenged network, provided this communication system can operate satisfactorily in this challenged network.

Another area of future work lies in providing the set of programming models presented in Section 2.4.3.1 for DTNs. DTNs, as for OppNets, require programming models that are inherently asynchronous on some levels (space, time or thread), as discussed in Section 2.4.3.1. However, DTNs are not characterized by opportunistic connectivity between nodes. Consequently, the current implementations of JOMS, JION and ADAM (that leverage DoDWAN) are not optimal to provide the set of programming models in DTNs, as their performance will be noticeably degraded for such environments. In order to optimize the performance of our middleware systems in the context of DTNs, they should rely DTN2 that can efficiently support DTNs.

7.2.3 Combination of Middleware Systems

Another area of future work lies in the combination of the various middleware systems presented in this dissertation. Such combination would allow these middleware systems to leverage each other so as to support some of their operations differently in order to be well-adapted with the dynamic nature of OppNets.

For example, ADAM could be leveraged by both JOMS and JION so as to allow them support their operations in a different way that is more adapted with the dynamic nature of OppNets, as discussed in Section 6.8. Similarly, JOMS could leverage the tuple space programming model provided by JION to support the message queue programming model differently so as to allow messages sent to a queue to be stored in the tuple space rather than being stored in the corresponding queue manager.

However, the combination of different middleware systems might cause some inconvenience. Indeed, the experimental results clearly show that these disruption-tolerant middleware systems take a while in order to perform their operations. Therefore, performing operations that leverage a disruption-tolerant middleware system might be time-consuming. However, the performance gain achieved through such combination could sometimes compensate the extra cost. More investigation is required to evaluate the cost/performance ratio of such combination.

7.2.4 Security

Security issues have not been addressed while designing the middleware systems in this dissertation. Among the different elements of information security proposed by Donn B. Parker in [119], we are here interested in *confidentiality*, *integrity* and *authenticity*. The lack of these security elements in middleware systems designed for OppNets may have serious consequences, since it would inherently foster mobile devices to behave individually, since they cannot trust other devices.

One of the main features of OppNets is the lack of an infrastructure, which could be considered as a critical aspect from a security standpoint. Indeed, the lack of infrastructure makes it difficult to apply conventional security mechanisms developed for traditional networks. Mobile devices form an OppNet without the help of any security infrastructure and without any priori trust relationship between themselves. Ensuring security in such a situation is not a trivial task. This task can be simplified by only allowing closed communities to form OppNets so that each member can trust the others. However, such a solution tends to go against the openness of OppNets.

In content-centric based systems, the decisions of forwarding are normally based on the content of messages. However, for security purposes, such information has to be encrypted. Therefore, a main challenge is to solve the conflict between security and opportunistic forwarding by allowing networking mechanisms to operate on encrypted data. Alternatively, messages can include encrypted content along with unencrypted metadata. Hence, the decisions of forwarding can be made based on the unencrypted metadata. Yet, for security purposes, relay nodes might not accept to relay messages with encrypted contents.

7.2.5 Simulation

OppNets are not yet commonly available outside research environments. Any experiment that sheds the light on what they would be used for, and how they should operate, would help research.

In this dissertation, several experiments were conducted in "real-life" conditions (i.e. with a small set of volunteers, using real distributed applications based on our systems). Each experiment was actually meant to serve as a proof-of-concept, to demonstrate that these middleware systems can indeed operate satisfactorily in an OppNets, using disruption-tolerant networking as a means to tolerate the absence of end-to-end connectivity in the network. Hence they are not sufficient to draw general insights on the different aspects of OppNets.

Performing real-world experiments of a middleware system in an OppNet permits to accurately capture system interactions and environment characteristics. However, extending the scope (in terms of population, duration, and geographical area) of a real experiment does not necessarily provide more significant results, as discussed in Section 6.7.1. Researchers basically relay on simulation that provides advantages in terms of scalability and repeatability. However, they commonly provide limited fidelity especially for OppNets [120].

After investigating the performance of the middleware systems presented in this dissertation in real-life conditions, the next step consists of performing simulations so as evaluate their performance in a large-scale network involving a larger population. In this regard, ADAM was tested using the MobSim simulator, a network simulator that investigates the performance of disruption-tolerant systems in a simulated OppNet.

Personal Publications

Journal articles

- Abdulkader Benchi, Pascale Launay, Frédéric Guidec. JMS for Opportunistic Networks. Ad Hoc Networks, Elsevier, 2015, 25 (part B), pp.359-369.
- Abdulkader Benchi, Pascale Launay, Frédéric Guidec. A P2P Tuple Space Implementation for Disconnected MANETs. Peer-to-Peer Networking and Applications, Springer, 2015, 8 (1), pp.87-102.

Conference papers

- Abdulkader Benchi, Pascale Launay, Frédéric Guidec. Solving Consensus in Opportunistic Networks. 2015 International Conference on Distributed Computing and Networking, Jan 2015, Goa, India. Proceedings of the 2015 International Conference on Distributed Computing and Networking, pp.1:1-1:10.
- Abdulkader Benchi, Pascale Launay, Frédéric Guidec. JION: A JavaSpaces Implementation for Opportunistic Networks. The Fourth International Conference on Future Computational Technologies and Applications, Jul 2012, Nice, France. pp.49-54.
- Abdulkader Benchi, Frédéric Guidec, Pascale Launay. A Message Service for Opportunistic Computing in Disconnected MANETs. 12th IFIP International Conference on Distributed Applications and Interoperable Systems, Jun 2012, Stockholm, Sweden. pp.118-131.

Workshop papers

- Abdulkader Benchi, Frédéric Guidec, Pascale Launay. JOMS: a Java Message Service Provider for Disconnected MANETs. 8th International Workshop on Heterogeneous Wireless Networks, Mar 2012, Fukuoka, Japan. pp.484-489.

Bibliography

[1] Jochen H. Schiller. *Mobile communications*. Addison-Wesley Longman, Incorporated, 2003.

[2] Changling Liu and Jörg Kaiser. A survey of mobile ad hoc network routing protocols. Technical report, University of Magdeburg, 2005.

[3] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. pages 27–34, New York, USA, 2003. ACM.

[4] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-Varying Graphs and Dynamic Networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, April 2012.

[5] Maurice J. Khabbaz, Assi Chadi M., and Fawaz Wissam F. Disruption-Tolerant Networking: a Comprehensive Survey on Recent Developments and Persisting Challenges. *IEEE Communications Surveys and Tutorials*, 14(2):607–640, 2012.

[6] Luciana Pelusi, Andrea Passarella, and Marco Conti. Opportunistic Networking: Data Forwarding in Disconnected Mobile Ad Hoc Networks. *IEEE Communications Magazine*, November 2006.

[7] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of ACM SIGCOMM03*, August 2003.

[8] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-tolerant networking: an approach to interplanetary internet. *IEEE Communications Magazine*, 41(6):128–136, 2003.

[9] Keith Scott. Disruption tolerant networking proxies for on-the-move tactical networks. In IEEE, editor, *IEEE Military Communications Conference (MILCOMM05)*, volume 5, 2005.

[10] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay-Tolerant Networking Architecture. IETF RFC 4838, April 2007.

[11] Anders Lindgren, Avri Doria, and Samo Grasic. Probabilistic Routing Protocol for Intermittently Connected Networks. In *Proceedings of the 4th International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2003)*, pages 233–244, Annapolis, MD, USA, June 2003. ACM.

[12] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Pocket switched networks: Real-world mobility and its consequences for opportunistic forwarding. Technical report, Technical Report UCAM-CL-TR-617, University of Cambridge, Computer Laboratory, 2005.

[13] Julien Haillot and Frederic Guidec. A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks. In *IEEE 22nd International Conference on Advanced Information Networking and Applications (AINA'08)*, pages 188–195, Okinawa, Japan, March 2008. IEEE CS.

[14] Thrasyvoulos Spyropoulos and Andreea Picu. *Opportunistic Routing*, pages 419–452. John Wiley and Sons, Inc., March 2013.

[15] Nilanjan Banerjee, Mark D. Corner, Don Towsley, and Brian N. Levine. Relays, base stations, and meshes: Enhancing mobile networks with infrastructure. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, MobiCom '08, pages 81–91. ACM, 2008.

[16] M. Conti and S. Giordano. Mobile Ad Hoc Networking: Milestones, Challenges, and New Research Directions. *Communications Magazine, IEEE*, 52(1):85–96, January 2014.

[17] Marco Conti, Silvia Giordano, Martin May, and Andrea Passarella. From Opportunistic Networks to Opportunistic Computing. *IEEE Communications Magazine*, 48(9):126–139, September 2010.

[18] Kamini Kamini and Rakesh Kumar. VANET Parameters and Applications: A Review. *Global Journal of Computer Science and Technology*, 10(7), 2010.

[19] Felipe J. Gil-Castiñeira, Francisco J. González-Castaño, and Laurent Franck. Extending Vehicular CAN Fieldbuses With Delay-Tolerant Networks. *IEEE Transactions on Industrial Electronics*, 55(9):3307–3314, 2008.

[20] N. Wisitpongphan, Fan Bai, P. Mudalige, and O.K. Tonguz. On the Routing Problem in Disconnected Vehicular Ad-hoc Networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2291–2295, May 2007.

[21] M. Abuelela, S. Olariu, and I. Stojmenovic. OPERA: Opportunistic packet relaying in disconnected Vehicular Ad Hoc Networks. In *Mobile Ad Hoc and Sensor Systems, 2008. MASS 2008. 5th IEEE International Conference on*, pages 285–294, 2008.

[22] Marcelo Gonçalves Rubinstein, Fehmi Ben Abdesslem, M Dias de Amorim, Sávio Rodrigues Cavalcanti, Rafael dos Santos Alves, Luís Henrique Maciel Kosmalski Costa, Otto Carlos Muniz Bandeira Duarte, and Miguel Elias Mitre Campista. Measuring the Capacity of In-Car to In-Car Vehicular Networks. *Communications Magazine, IEEE*, 47(11):128–136, November 2009.

[23] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107. ACM Press, 2002.

[24] Tara Small and Zygmunt J. Haas. The Shared Wireless Infostation Model: a New Ad Hoc Networking Paradigm (or Where There is a Whale, There is a Way). In *MobiHoc*, pages 233–244. ACM, 2003.

[25] Alex Pentland, Richard Fletcher, and Amir Hasson. Daknet: Rethinking connectivity in developing nations. *IEEE Computer*, 1(37):78–83, January 2004.

[26] Hervé Ntareme, Marco Zennaro, and Björn Pehrson. Delay Tolerant Network on Smartphones: Applications for Communication Challenged Areas. In *Proceedings of the 3rd Extreme Conference on Communication: The Amazon Expedition*, ExtremeCom '11, pages 14:1–14:6. ACM, 2011.

[27] Vinícius F. S. Mota, Felipe D. Cunha, Daniel F. Macedo, José M. S. Nogueira, and Antonio A. F. Loureiro. Protocols, mobility models and tools in opportunistic networks: A survey. *Computer Communications*, March 2014.

[28] James P. G. Sterbenz, David Hutchison, Egemen K. Çetinkaya, Abdul Jabbar, Justin P. Rohrer, Marcus Schöller, and Paul Smith. Resilience and Survivability in Communication Networks: Strategies, Principles, and Survey of Disciplines, June 2010.

[29] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design (5th Edition)*. Addison Wesley, 2011.

[30] Gruia-Catalin Roman, Amy L. Murphy, and Gian Pietro Picco. A software engineering perspective on mobility. In *Future of Software Engineering*. ACM Press, 1999.

[31] Wolfgang Emmerich. *Engineering Distributed Objects*. Wiley, 1 edition, Jun 2000.

[32] Stephen Paul Wade. An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments, 1999.

[33] Alan Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1998.

[34] Esmond Pitt and Kathy McNiff. *Java.rmi: The Remote Method Invocation Guide*. Pearson Education, Jul 2001.

[35] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[36] Vlada Matena and Mark Hapner. Enterprise JavaBeans Specification, v1.1. Technical report, Sun Microsystems, Palo Alto, CA, 1999.

[37] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.

[38] Group Object Management. Corba component model 4.0 specification. Technical Report Version 4.0, Object Management Group, April 2006.

[39] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *DIS, Universita di Roma La Sapienza, Tech. Rep*, 2005.

[40] Timothy Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time corba event service. In *in Proceedings of OOPSLA '97, (Atlanta, GA), ACM*, pages 184–199. ACM, 1997.

[41] Mark Hapner, Rich Burridge, and Rahul Sharma. Java Message Service, Version 1.1, April 2002.

[42] Saida Davies and Peter Broadhurst. *WebSphere MQ V6 Fundamentals*. IBM, 2005.

[43] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 1 edition, October 2003.

[44] Apache Software Foundation. Apache Axis2/Java - Next Generation Web Services, July 2009.

[45] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985.

[46] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces(TM) Principles, Patterns, and Practice*. Prentice Hall, June 1999.

[47] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[48] Stephen Paul Wade. *An Investigation Into the Use of the Tuple Space Paradigm in Mobile Computing Environments*. University of Lancaster, 1999.

[49] Yves Mahéo and Romeo Said. Service Invocation over Content-Based Communication in Disconnected Mobile Ad Hoc Networks. In *24th International Conference on Advanced Information Networking and Applications (AINA'10)*, pages 503–510, Perth, Australia, April 2010. IEEE CS.

[50] Yves Mahéo, Romeo Said, and Frédéric Guidec. Middleware Support for Delay-Tolerant Service Provision in Disconnected Mobile Ad Hoc Networks. In *Workshop on Java and Components for Parallelism, Distribution and Concurrency at IPDPS'08*, Miami, FL, USA, April 2008. IEEE CS.

[51] Luc Hogie, Pascal Bouvry, and Frédéric Guinand. An Overview of MANETs Simulation. *Electronic Notes in Theoretical Computer Science*, 150(1):81 – 101, 2006.

[52] Sagar A. Tamhane, Mohan Kumar, Andrea Passarella, and Marco Conti. Service composition in opportunistic networks. In *Proceedings of the 2012 IEEE International Conference on Green Computing and Communications*, GREENCOM '12, pages 285–292. IEEE Computer Society, 2012.

[53] Marco Conti, Emanuel Marzini, Davide Mascitti, Andrea Passarella, and Laura Ricci. Service selection and composition in opportunistic networks. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 1565–1572. IEEE, July 2013.

[54] Mikko Pitkänen, Teemu Kärkkäinen, Jörg Ott, Marco Conti, Andrea Passarella, Silvia Giordano, Daniele Puccinelli, Franck Legendre, Sacha Trifunovic, Karin Hummel, et al. SCAMPI: Service Platform for Social Aware Mobile and Pervasive Computing. *ACM SIGCOMM Computer Communication Review*, 42(4):503–508, 2012.

[55] Julien Haillot and Frédéric Guidec. A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks. *Journal of Mobile Information Systems*, 6(2):123–154, 2010.

[56] Amin Vahdat and David Becker. Epidemic Routing for Partially Connected Ad Hoc Networks. Technical report, Duke University, April 2000.

[57] Anwitaman Datta, Silvia Quarteroni, and Karl Aberer. Autonomous Gossiping: a Self-Organizing Epidemic Algorithm for Selective Information Dissemination in Mobile Ad Hoc Networks. In *International Conference on Semantics of a Networked World*, number 3226 in LNCS, pages 126–143, Paris, France, June 2004.

[58] Anders Lindgren, Avri Doria, and Olov Schelen. Probabilistic Routing in Intermittently Connected Networks. In *Proceedings of the 1st International Workshop on Service Assurance with Partial and Intermittent Resources*, Fortaleza, Brazil, August 2004.

[59] Hoang Anh Nguyen, Silvia Giordano, and Alessandro Puiatti. Probabilistic Routing Protocol for Intermittently Connected Mobile Ad hoc Network (PROPICMAN). In *International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–6, Helsinky, Finland, June 2007. IEEE CS.

[60] Zhensheng Zhang. Routing in Intermittently Connected Mobile Ad Hoc Networks and Delay Tolerant Networks: Overview and Challenges. *IEEE Communications Surveys and Tutorials*, 8(1):24–37, January 2006.

[61] Valerio Arnaboldi, Marco Conti, and Franca Delmastro. CAMEO: A Novel Context-Aware Middleware for Opportunistic Mobile Social Networks. *Pervasive and Mobile Computing*, 2013.

[62] Chiara Boldrini, Marco Conti, and Andrea Passarella. Context and Resource Awareness in Opportunistic Network Data Dissemination. In *The Second IEEE WoWMoM Workshop on Autonomic and Opportunistic Communications*, Newport Beach, CA, USA, June 2008.

[63] Rui Filipe Pedro Quelhas. *Improving Opportunistic with Social Context Communications*. PhD thesis, Universidade do Minho, Escola de Engenharia, October 2011.

[64] Mirco Musolesi, Ben Hui, Cecilia Mascolo, and Jon Crowcroft. Writing on the Clean Slate: Implementing a Socially-Aware Protocol in Haggle. In *IEEE International Workshop on Autonomic and Opportunistic Communications*, Newport Beach, CA, jun 2008.

[65] James Scott, Pan Hui, Jon Crowcroft, and Christophe Diot. Haggle: a Networking Architecture Designed Around Mobile Users. In *Proceedings of the 2006 IFIP Conference on Wireless on Demand Network Systems and Services*, January 2006.

[66] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Impact of Human Mobility on Opportunistic Forwarding Algorithms. *IEEE Transactions on Mobile Computing*, 6(6):606–620, jun 2007.

[67] Pan Hui, Jon Crowcroft, and Eiko Yoneki. BUBBLE Rap: Social Based Forwarding in Delay Tolerant Networks. In *9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 241–250, Hong Kong, China, may 2008. ACM.

[68] Pan Hui, Eiko Yoneki, Shu-Yan Chan, and Jon Crowcroft. Distributed Community Detection in Delay Tolerant Networks. In *Sigcomm Workshop MobiArch*, Kyoto, Japan, aug 2007.

[69] Ling-Jyh Chen, Chen-Hung Yu, Cheng-Long Tseng, Hao-Hua Chu, and Cheng-Fu Chou. A Content-Centric Framework for Effective Data Dissemination in Opportunistic Networks. *IEEE Journal of Selected Areas in Communications*, 2008.

[70] Khaled A. Harras, Kevin C. Almeroth, and Elisabeth M. Belding-Royer. Delay Tolerant Mobile Networks (DTMNs): Controlled Flooding in Sparse Mobile Networks. In *IFIP Networking Conference, Waterloo, Ontario, Canada*, May 2005.

[71] Jonah P Tower and Thomas DC Little. A Proposed Scheme for Epidemic Routing With Active Curing for Opportunistic Networks. In *2008 22nd International Workshops on Advanced Information Networking and Applications (AINA Workshops)*, pages 1696–1701. IEEE, 2008.

[72] Erik Nordström, Christian Rohner, and Per Gunningberg. Haggle: Opportunistic Mobile Content Sharing Using Search. *Computer Communications*, 48(0):121 – 132, 2014.

[73] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic Routing in Intermittently Connected Networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(3):19–20, July 2003.

[74] Elizabeth M. Daly and Mads Haahr. Social Network Analysis for Routing in Disconnected Delay-tolerant MANETs. In *Proceedings of the 8th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '07, pages 32–40. ACM, 2007.

[75] Antonio Carzaniga and Alexander L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in LNCS, pages 59–68, Scottsdale, Arizona, October 2001. Springer-Verlag.

[76] Michael Meisel, Vasileios Pappas, and Lixia Zhang. Ad hoc networking via named data. In *Proceedings of the Fifth ACM International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '10, pages 3–8, New York, NY, USA, 2010. ACM.

[77] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In *Advanced lectures on networking*, pages 20–58. Springer, 2002.

[78] Wolfgang Emmerich. Software engineering and middleware: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 117–129, New York, NY, USA, 2000. ACM.

[79] Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. *Openjms*. VDM Publishing, November 2010.

[80] Rosanna Lee and Scott Seligman. *JNDI API Tutorial and Reference: Building Directory-Enabled Java Applications*. Addison-Wesley, Reading, USA, 2000.

[81] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. EMMA: Epidemic Messaging Middleware for Ad Hoc Networks. *Personal and Ubiquitous Computing*, 10(1):28–36, August 2005.

[82] Muhammad Yasir Malik. Power consumption analysis of a modern smartphone. *arXiv preprint arXiv:1212.1896*, 2012.

[83] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 271–285. USENIX Association, 2010.

[84] Wei Lin and Joshua A Wise. Precise power characterization of modern android devices, 2010.

[85] Salem Hadim, Jameela Al-Jaroodi, and Nader Mohamed. Trends in Middleware for Mobile Ad Hoc Networks. *Journal of Communications*, 1(4), July 2006.

[86] Einar Vollset, Dave Ingham, and Paul Ezhilchelvan. JMS on Mobile Ad Hoc Networks. *In Personal Wireless Communications (PWC)*, pages 40–52, 2003.

[87] Haitian Chen, Liang Chen, Kavitha Gupta, Christos Savvidis, and Wai Git Teo. Reliable Asynchronous Middleware for Mobile Ad Hoc Networks. September 2008.

[88] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[89] Gruia-Catalin Roman, Amy L. Murphy, and Gian Pietro Picco. Coordination and mobility. In *Coordination of Internet Agents: Models, Technologies, and Applications*, page 254–273. Springer, 1999.

[90] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Tuple Space Middleware for Wireless Networks. In Benoit Garbinato, Hugo Miranda, and Louis Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 245–264. Springer Press, 2009. Invited contribution.

[91] E.F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 253 –259, may-1 jun 1990.

[92] Java SE Documentation. Concurrency Utilities. Technical report, 2011.

[93] Internet Engineering Task Force. vCard format specification. http://tools.ietf.org/html/rfc6350.

[94] Alan Kaminsky and C. Bondada. Tuple board: A new distributed computing paradigm for mobile ad hoc networks. *First Annual Conference on Computing and Information Sciences*, pages 5–7, 2005.

[95] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15:279–328, July 2006.

[96] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents*, page 50–61, London, UK, 1997. Springer-Verlag.

[97] Chien-Liang Fok, Gruia-Catalin Roman, and Gregory Hackmann. A lightweight coordination middleware for mobile computing. *Coordination Models and Languages*, pages 135–151, 2004.

[98] Gruia-Catalin Roman, Radu Handorean, and Rohan Sen. Tuple space coordination across space and time. In *COORDINATION*, pages 266–280, 2006.

[99] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems, 2002. Slide presentation can be found at the following link: http://www.cs.rit.edu/~ark/20021107/slide01.html.

[100] Juerg Kohlas, Bertrand Meyer, and André Schiper. *Dependable Systems: Software, Computing, Networks: Research Results of the DICS Program*. LNCS sublibrary: Programming and software engineering. Springer, 2006.

[101] Fatemeh Borran, Ravi Prakash, and André Schiper. Extending Paxos/LastVoting with an Adequate Communication Layer for Wireless Ad Hoc Networks. In *2008 Symposium on Reliable Distributed Systems*, page 227–236. IEEE, 2008.

[102] Bernadette Charron-Bost and André Schiper. The Heard-Of Model: Computing in Distributed Systems With Benign Faults. *Distributed Computing*, 22(1):49–71, 2009.

[103] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus With One Faulty Process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[104] Francisco Vilar Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in One Communication Step. In *Parallel Computing Technologies*, volume 2127 of *LNCS*, pages 42–50. Springer, 2001.

[105] Yves Mahéo, Nicolas Le Sommer, Pascale Launay, Frédéric Guidec, and Mario Dragone. Beyond Opportunistic Networking Protocols: a Disruption-Tolerant Application Suite for Disconnected MANETs. In *4th Extreme Conference on Communication (ExtremeCom'12)*, pages 1–6, Zurich, Switzerland, March 2012.

[106] Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, Seong Joon Kim, and Song Chong. On the levy-walk nature of human mobility. *Networking, IEEE/ACM Transactions on*, 19(3):630–643, June 2011.

[107] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Impact of human mobility on opportunistic forwarding algorithms. *Mobile Computing, IEEE Transactions on*, 6(6):606–620, June 2007.

[108] Einar W Vollset and Paul D Ezhilchelvan. Design and Performance-Study of Crash-Tolerant Protocols for Broadcasting and Reaching Consensus in MANETs. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, volume 0, page 166–178, Washington, DC, USA, 2005. IEEE Computer Society.

[109] Weigang Wu, Jiannong Cao, Jin Yang, and Michel Raynal. Design and Performance Evaluation of Efficient Consensus Protocols for Mobile Ad Hoc Networks. *IEEE Transactions on Computers*, 56(8):1055–1070, 2007.

[110] David Cavin, Yoav Sasson, and André Schiper. Consensus with Unknown Participants or Fundamental Self-Organization. In *ADHOC-NOW*, volume 3158 of *LNCS*, pages 135–148. Springer, 2004.

[111] David Cavin, Yoav Sasson, and André Schiper. Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes. Technical report, 2005.

[112] Fabiola Greve and Sébastien Tixeuil. Knowledge Connectivity vs. Synchrony Requirements for Fault-Tolerant Agreement in Unknown Networks. In *Dependable Systems and Networks, 2007. 37th Annual IEEE/IFIP International Conference on*, pages 82–91. IEEE, June 2007.

[113] John Augustine, Gopal Pandurangan, and Peter Robinson. Fast Byzantine Agreement in Dynamic Networks. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 74–83, New York, NY, USA, 2013. ACM.

[114] Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. Highly Dynamic Distributed Computing with Byzantine Failures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 176–183. ACM, 2013.

[115] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *In Proceedings of The Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, 1994.

[116] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[117] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing, Second Edition*. Morgan Kaufmann, Burlington, Mass. u.a., 2 edition edition, June 2009.

[118] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[119] Donn B. Parker. *Fighting Computer Crime - a New Framework for Protecting Information.* Wiley, 1998.

[120] Nikodin Ristanovic, George Theodorakopoulos, and Jean-Yves Le Boudec. Traps and pitfalls of using contact traces in performance studies of opportunistic networks. In *INFOCOM*, pages 1377–1385, 2012.

Résumé

Les réseaux mobiles opportunistes (ou OppNets, pour Opportunistic Networks) constituent une solution séduisante pour compléter les réseaux fixes d'infrastructure, voire compenser leur absence dans des zones sinistrées ou défavorisées.
Les recherches menées ces dernières années ont principalement visé à permettre les transmissions dans les OppNets, mais ceci ne peut être qu'un premier pas vers une réelle exploitation de tels environnements contraints. L'informatique opportuniste (Opportunistic Computing) dépasse le cadre des seules transmissions, et introduit un nouveau paradigme d'exécution de tâches collaboratives dans de tels environnements.
Dans ce domaine qu'est l'informatique opportuniste, la conception, la mise en œuvre et le déploiement d'applications distribuées sont des objectifs majeurs. Une application pour OppNet doit pouvoir fonctionner et assurer un niveau de service satisfaisant, tout en supportant les diverses contraintes propres aux OppNets, telles qu'une connectivité fluctuante, un partitionnement chronique du réseau, de longs délais de transmissions, de fréquents échecs de transmission, et des équipements hétérogènes offrant des ressources limitées.
La complexité et le coût du développement d'applications pour OppNets peuvent être réduits de manière significative en utilisant des modèles de programmation appropriés. De tels modèles peuvent être fournis par des systèmes intergiciels capables de supporter de manière transparente les contraintes évoquées plus haut.
Le travail rapporté dans ce mémoire a porté sur l'étude des contraintes inhérentes aux OppNets, et sur la proposition de solutions appropriées. Parmi les modèles de programmation usuels, certains ont été identifiés comme pouvant être utilisés dans le cadre des OppNets. Sur la base de ces divers modèles de programmation, des systèmes intergiciels opportunistes ont été mis en œuvre. Ces systèmes supportent respectivement le modèle de messagerie distribuée (sur la base de files d'attentes et de "topics"), le modèle du tuple-space, et la résolution de consensus. Des implémentations complètes ont été réalisées, et le code source est distribué sous licence GPL (GNU General Public License). Ces systèmes ont été évalués par le biais d'expérimentations menées en conditions réelles et par simulation.

Abstract

Opportunistic networks (OppNets) constitute an appealing solution to complement fixed network infrastructures – or make up for the lack thereof– in challenged areas. Researches in the last few years have mostly addressed the problem of supporting networking in OppNets, yet this can only be a first step towards getting real benefit from these networks. Opportunistic computing goes beyond the concept of opportunistic networking, and provides a new paradigm to enable collaborative computing tasks in such environments.
In the realm of opportunistic computing, properly designing, implementing and deploying distributed applications are important tasks. An OppNet-dedicated application must be able to operate and maintain an acceptable level of service while addressing the many problems that can occur in these networks, such as disconnections, partitioning, long transmission delays, transmission failures, resource constraints, frequent changes in topology, and heterogeneous devices.
Much of the complexity and cost of building OppNet-dedicated applications can be alleviated by the use of high-level programming models. Such models can be supported by middleware systems capable of transparently addressing all the above-mentioned problems.
The work reported in this dissertation focused on providing insight into the fundamental problems posed by OppNets, so as to analyze and solve the problems faced by application developers while dealing with these environments. The research focused on identifying well-known high-level programming models that can be satisfactorily implemented for OppNets, and that can prove useful for application developers. In order to demonstrate the feasibility of application development for OppNets, while assessing the benefits brought about by carefully designed middleware systems, a couple of such systems have been designed, implemented, and evaluated as part of this work.
These middleware systems respectively support distributed messaging (through message queues and topics), the tuple-space model, and consensus solving in OppNets. They are supplemented with fully-functional implementations, that can be used in real settings, and that are all distributed under the terms of the GNU General Public License (GPL). Real-life experiments and simulations have been realized so as to evaluate the effectiveness and efficiency of these systems in real conditions.

n d'ordre : *****

Université de Bretagne Sud
Centre d'Enseignement et de Recherche Y. Coppens - rue Yves Mainguy - 56000 VANNES
Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70